Spatial Isolation against Logical Cache-based Side-Channel Attacks in Many-Core Architectures

Maria Méndez Real

UNIVERSITE
BRETAGNE
LOIRE

THÈSE / UNIVERSITÉ BRETAGNE-SUD
sous le sceau de l'Université Bretagne-Loire

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ BRETAGNE-SUD

Mention : STIC
École doctorale : SICMA

Présentée par
Maria Méndez Real

Préparée à l'unité mixte de recherche
(n° 6285)
Laboratoire des Sciences et Techniques de l'Information,
de la Communication et de la Connaissance

Spatial Isolation against Logical Cache-based Side-Channel Attacks in Many-Core Architectures

Thèse soutenue le 20 septembre 2017
devant le jury composé de :

Nele Mentens
Associate Professor, KU Leuven / Reviewer
Eduardo De La Torre
Associate Professor, Universidad Politécnica de Madrid (UPM) / Reviewer
Gilles Sassatelli
CNRS Research Director, Université de Montpellier / Reviewer
Lilian Bossuet
Professor, Université Jean Monnet / Examiner
Khurram Bhatti
Assistant Professor, Technology University (TU) / Examiner
Diana Goehringer
Professor, Technische Universitaet Dresden, Germany / Examiner
Vianney Lapotre
Associate Professor, Université de Bretagne-Sud (UBS) / Thesis Co-Adviser
Guy Gogniat
Professor, Université de Bretagne-Sud (UBS) / Thesis Adviser

Résumé

L'évolution technologique ainsi que l'augmentation incessante de la puissance de calcul requise par les applications font des architectures "many-core" la nouvelle tendance dans la conception des processeurs. Ces architectures sont composées d'un grand nombre de ressources de calcul (des centaines ou davantage) ce qui offre du parallélisme massif et un niveau de performance très élevé. En effet, les architectures many-core permettent d'exécuter en parallèle un grand nombre d'applications, venant d'origines diverses et de niveaux de sensibilité et de confiance différents, tout en partageant des ressources physiques telles que des ressources de calcul, de mémoire et de communication.

Cependant, ce partage de ressources introduit également des vulnérabilités importantes en termes de sécurité. En particulier, les applications sensibles partageant des mémoires cache avec d'autres applications, potentiellement malveillantes, sont vulnérables à des attaques logiques de type canaux cachés basées sur le cache. Ces attaques, permettent à des applications non privilégiées d'accéder à des informations secrètes sensibles appartenant à d'autres applications et cela malgré des méthodes de partitionnement existantes telles que la protection de la mémoire et la virtualisation.

Alors que d'importants efforts ont été faits afin de développer des contremesures à ces attaques sur des architectures multicoeurs, ces solutions n'ont pas été originellement conçues pour des architectures many-core récemment apparues et nécessitent d'être évaluées et/ou revisitées afin d'être applicables et efficaces pour ces nouvelles technologies.

Dans ce travail de thèse, nous proposons d'étendre les services du système d'exploitation avec des mécanismes de déploiement d'applications et d'allocation de ressources afin de protéger les applications s'exécutant sur des architectures many-core contre les attaques logiques basées sur le cache. Plusieurs stratégies de déploiement sont proposées et comparées à travers différents indicateurs de performance. Ces contributions ont été implémentées et évaluées par prototypage virtuel basé sur SystemC et sur la technologie "Open Virtual Platforms" (OVP).

Abstract

The technological evolution and the increasing application performance demand have made of many-core architectures the new trend in processor design. These architectures are composed of a large number of processing resources (hundreds or more) providing massive parallelism and high performance. Many-core architectures allow indeed a wide number of applications coming from different sources, with a different level of sensitivity and trust, to be executed in parallel, sharing physical resources such as computation, memory and communication infrastructure.

However, this resource sharing introduces important security vulnerabilities. In particular, sensitive applications sharing cache memory with potentially malicious applications are vulnerable to logical cache-based side-channel attacks. These attacks allow an unprivileged application to access sensitive information manipulated by other applications despite partitioning methods such as memory protection and virtualization.

While a lot of efforts on countering these attacks on multi-core architectures have been done, these have not been designed for recently emerged many-core architectures and require to be evaluated, and/or revisited in order to be practical for these new technologies.

In this thesis work, we propose to enhance the operating system services with security-aware application deployment and resource allocation mechanisms in order to protect sensitive applications against cached-based attacks. Different application deployment strategies allowing spatial isolation are proposed and compared in terms of several performance indicators. Our proposal is evaluated through virtual prototyping based on SystemC and Open Virtual Platforms (OVP) technology.

Remerciements

Ce manuscrit présente des travaux que j'ai effectués dans le cadre de ma thèse durant trois ans au laboratoire Lab-STICC. Je tiens à remercier les personnes de ce laboratoire qui ont permis à ce travail d'être réalisé dans des bonnes conditions et dans une bonne ambiance. Aussi, je remercie les personnes du laboratoire IETR qui m'ont permis de finaliser la préparation de ma soutenance de thèse dans des bonnes conditions.

Je remercie tous les membres du jury, mes rapporteurs Nele Mentens, Eduardo De La Torre et Gilles Sassatelli, mes examinateurs Diana Goehringer et Khurram Bhatti ainsi que Lilian Bossuet, examinateur et président de mon jury de thèse. Je les remercie d'avoir participé à la soutenance de ma thèse, de s'y être investi de manière aussi enthousiaste et d'avoir montré leur intérêt dans mes travaux de thèse. Je les remercie également pour leurs commentaires, leurs questions et leurs conseils.

Je remercie chaleureusement mon directeur et encadrant de thèse Guy Gogniat et mon encadrant Vianney Lapôtre. Merci de toujours avoir réussi à trouver du temps pour moi malgré vos emplois du temps serrés, de nos discussions, de votre encouragement et conseils, ainsi que et de la confiance dont vous m'avez toujours témoignée. Travailler avec vous a été un vrai plaisir et j'espère que nous aurons l'occasion de collaborer ensemble à nouveau.

Je remercie également tous les partenaires du projet TSUNAMY ainsi que l'équipe MCA de l'Université de la Ruhr à Bochum des collaborations que nous avons pu mener lors de ma thèse.

Aussi, je remercie mes collègues et mes amis au sein du laboratoire dont la liste est longue. Merci à tous de votre soutien, de votre aide et de votre bonne humeur.

Finalement je remercie ma famille proche et mon mari et meilleur ami Baptiste Goupille–Lescar de m'avoir toujours rassurée et encouragée.

Contents

List of Figures v

List of Tables ix

List of Algorithms xi

1 Introduction and context 1

1.1 Introduction . . . 1

1.1.1 TSUNAMY project . . . 3

1.2 Context . . . 5

1.2.1 Considered system . . . 6

1.2.2 Attacks characterization and threat model . . . 8

1.2.3 Threat model considered in this thesis work . . . 11

1.2.4 Introduction to logical cache-based attacks . . . 12

1.3 Contributions . . . 18

1.4 Organization of the manuscript . . . 20

2 State of the art 23

2.1 Software and hardware cache-based attacks countermeasures . . . 23

2.1.1 Software cache-based attacks countermeasures . . . 24

2.1.2 Hardware cache-based attacks countermeasures . . . 27

2.2 Logical and physical isolation . . . 33

2.3 Discussion . . . 40

3 Spatial isolation 43

3.1 Spatial isolation . . . 43

3.2 Different deployment strategies for spatial isolation . . . 46
3.2.1 Static size secure zone approach . . . 46
3.2.2 Fully dynamic size secure zone approach . . . 51
3.2.3 Trade-off strategies combining both, static and dynamic approaches 53
3.3 Summary of the proposed deployment strategies . . . 57
3.4 Extension of the kernel services for spatial isolation implementation . . . 58
3.4.1 Threat model and assumptions . . . 58
3.4.2 ALMOS kernel services and integration of spatial isolation . . . 60
3.5 Summary of the extensions of the kernel services . . . 69
3.6 Conclusion . . . 71

4 Extension and exploration of MPSoCSim 73
4.1 Motivation . . . 74
4.2 MPSoCSim . . . 77
4.2.1 Imperas/Open Virtual Platforms (OVP) technology . . . 77
4.2.2 Network-on-Chip (NoC) simulator . . . 77
4.3 Validation . . . 80
4.3.1 Hardware implementation . . . 80
4.3.2 Experimental protocol . . . 81
4.3.3 Evaluation results . . . 82
4.4 MPSoCSim extension . . . 83
4.4.1 Clustering the architecture . . . 83
4.4.2 Dynamic execution capabilities . . . 86
4.5 Exploration with the extended MPSoCSim version . . . 89
4.5.1 Exploration protocol . . . 89
4.5.2 MPSoCSim available results . . . 90
4.5.3 Exploration results . . . 91
4.6 Conclusion . . . 94

5 Spatial isolation evaluation through virtual prototyping 97
5.1 Experimental protocol . . . 98
5.2 Cache attacks vulnerability case study . . . 100
5.3 Deployment strategies comparison results . . . 102
5.3.1 Considered deployment and execution scenarios . . . 102
5.3.2 Results organization . . . 103

5.3.3 Comparison according to each performance indicator 104
5.3.4 Comparison according to each execution scenario 108
5.3.5 Comparison summary 110
5.4 Conclusion 116

6 Conclusion and future work **117**
6.1 Summary 117
6.2 Spatial isolation discussion 118
6.2.1 Back to the state-of-the-art 118
6.2.2 Further attacks and spatial isolation capabilities 120
6.3 Possible improvements and leads for future work 121
6.4 Conclusion 128

Glossary **129**

List of publications and presentations **131**

Bibliography **135**

List of Figures

1.1 Moore's law 2
1.2 TSUNAMY context 4
1.3 Overview of the considered system 7
1.4 Principle of PRIME+PROBE [29] attack 15

2.1 Cache partitions using page coloring 28
2.2 Attack on NoC communication 29
2.3 A logical view of the RPcache permutation [69] 31
2.4 Cache access handling in RPcache [69] 32
2.5 Memory Management Unit (MMU) 34
2.6 Modes in ARM core with TrustZone extensions [81] 36
2.7 Dynamic security domains [83] 38
2.8 Extension of SCC for resource isolation [84] 39

3.1 Principle of spatial isolation 45
3.2 Creating a Secure Zone 47
3.3 Overview of the creation of a static size secure zone flow 49
3.4 Exploration of clusters in secure zone creation algorithms 51
3.5 Overview of the dynamic approach with a guaranteed non-optimized secure zone size approach 55
3.6 Overview of the resource reservation approach 57
3.7 Overview of the kernel monitoring structure 61
3.8 Overview of original kernel application mapping algorithm and its extension in dashed lines (double line nodes are replaced by dashed line ones) 63
3.9 Overview of the kernel functioning when mapping a new application 64

3.10 Overview of original kernel task mapping algorithm and its extension in dashed lines (double line nodes are replaced by dashed line ones) 67

4.1 2D-Mesh NoC for MPSoCSim validation 78
4.2 MPSocSim router structure 79
4.3 MPSocSim simulator overview 80
4.4 Tiled matrix multiplication [90] 82
4.5 Comparison between MPSoCSim and the hardware implementation [90] . . 83
4.6 Overview of the extended MPSoCSim 85
4.7 Considered platform instantiation 86
4.8 OVP ARM processor instantiation parameters 87
4.9 OVP MicroBlaze processor instantiation parameters 87
4.10 First possible case of dynamic execution simulation 88
4.11 Second possible case of dynamic execution simulation 89
4.12 **Elapsed time** (in msec.) for different sizes of matrix multiplications on 2×2 and 4×4 NoC clustered architectures (12 MicroBlazes and 1 ARM, and 60 MicroBlazes and 1 ARM respectively) 91
4.13 **Total simulated execution time** (in msec.) for different sizes of matrix multiplications on a 2×2 and 4×4 NoC clustered architectures (12 MicroBlazes and 1 ARM, and 60 MicroBlazes and 1 ARM respectively) 92
4.14 **Total simulated instructions** for the execution of different sizes of matrix multiplications on 2×2 and 4×4 NoC clustered architectures (12 MicroBlazes and 1 ARM, and 60 MicroBlazes and 1 ARM respectively) 93
4.15 Evaluation of the **simulated instructions per processor** for different sizes of matrix multiplications on a 2×2 architecture (12 MicroBlazes and 1 ARM) 94

5.1 *Total execution time* overhead (in percentage) compared to the Baseline strategy for each couple of secure zone deployment strategy and execution scenario 105
5.2 Overhead on the average *execution time of isolated applications*, in terms of percentage of the average application execution time in the Baseline strategy for each couple of secure zone deployment strategy and execution scenario . 107

5.3 Overhead on the average *execution time of non-isolated applications*, in terms of percentage of the average application execution time in the Baseline strategy for each couple of secure zone deployment strategy and execution scenario 108
5.4 *Time spent on the trusted manager kernel services* impacted by the secure-enable mechanisms in terms of percentage compared to the Baseline strategy for each couple of secure zone deployment strategy and execution scenario . 109
5.5 Overhead on the average *waiting time* before the deployment of isolated applications, in terms of percentage compared to the Baseline strategy, for each couple of secure zone deployment strategy and execution scenario . . . 110
5.6 Comparison of different strategies - Exec. scenario **a**. Values are presented in terms of induced overhead in percentage with respect of the baseline value. **Arranged scale**: the closest to the chart, the better. Scale: 1 division : 10% 112
5.7 Comparison of different strategies - Exec. scenario **b**. Values are presented in terms of induced overhead in percentage with respect of the baseline value. **Arranged scale**: the closest to the chart, the better. Scale: 1 division : 20% 113
5.8 Comparison of different strategies - Exec. scenario **c**. Values are presented in terms of induced overhead in percentage with respect of the baseline value. **Arranged scale**: the closest to the chart, the better. Scale: 1 division : 20% 114

6.1 Dynamic memory-to-cache mapping to reduce under-utilization or resources 126

List of Tables

1.1 Categories of logical attacks according to the threaten security property . . 11

2.1 Comparison of state-of-the-art countermeasures 41

3.1 Summary of the different proposed strategies 59
3.2 Summary of the extensions of the kernel services 70

4.1 Overview of the existing simulators supporting NoC infrastructure 76
4.2 System parameters used for evaluation in MPSoCSim 81

5.1 Cache-based attacks vulnerability in the Baseline strategy where the average resources utilization rate is 77% 101
5.2 *Resources utilization rate* within secure zone(s) as well as in total (referred to as *Secure zone* and *Total* in the table) for each couple of secure zone deployment strategy and execution scenario. For each *SZ* and *Total* columns, the best and worst resources utilization rates are highlighted in light gray and dark gray respectively. The *resources utilization rate* in the Baseline strategy is 77% 111
5.3 Selecting the deployment strategy summary 115

List of Algorithms

1 Creating a fixed secure zone 50
2 Creating a fully dynamic secure zone 52
3 Extending a dynamic size Secure Zone 53

Chapter 1

Introduction and context

Chapter contents

1.1 Introduction 1
1.1.1 TSUNAMY project 3
1.2 Context 5
1.2.1 Considered system 6
1.2.2 Attacks characterization and threat model 8
1.2.3 Threat model considered in this thesis work 11
1.2.4 Introduction to logical cache-based attacks 12
1.3 Contributions 18
1.4 Organization of the manuscript 20

In this chapter, a general introduction of this thesis work including the TSUNAMY project presentation, is first given. After, the system and the associated threat model are presented. Contributions are then presented and the organization of this manuscript is explained.

1.1 Introduction

The technological evolution has made possible the constant increase of transistor density. This evolution has been quantified by Gordon Moore, Intel co-founder, in 1965, by the empirical so called Moore's law. This law states that *the number of transistors per square inch on integrated circuits will double every year* [1]. Indeed, since that time, the number of transistors has doubled every 18 months approximately (Figure 1.1).

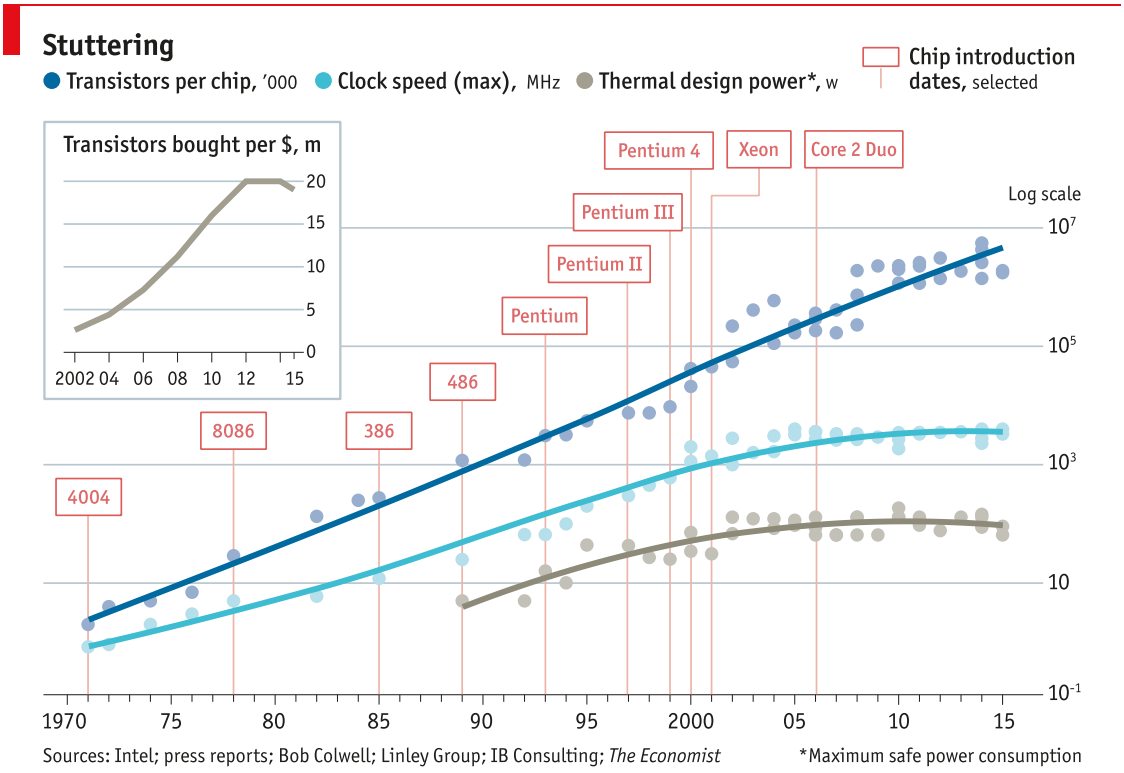


Figure 1.1 – Moore's law

Reducing the technology size makes possible to double the number of transistors but also to reduce the distance between them on the circuit. Consequently, circuits are more complex and the processing frequency is increased. However, the frequency improvement penalizes power consumption, which in turn generates heat, and increases leakage current. This trend introduces important issues; more advanced cooling is required, Thermal Design Power (TDP) constrains the amount of circuitry of the chip that can be powered-on at the nominal operating voltage (*dark silicon*), the decreasing distance between transistors entails important capacitance effects (*cross talk capacitance*), reducing the reliability of the system, and the circuit longevity is shortened.

A more cost-efficient alternative is to replicate multiple processing cores on a single die. These work in parallel, share memory and are connected via an on-chip bus. Following this trend, many-core architectures have been more recently emerged. These architectures, in contrast to multi-core, are composed of hundreds (or more) simpler and very efficient processing cores. Many-core architectures include private and shared memory, can encom-

pass heterogeneous processing resources offering great flexibility. Finally, these improve the performance of parallel computing applications as they offer a wide number of computing resources and optimize the communication between cores. For this purpose, applications running in parallel share physical resources for computation, data storage and communication. Existing commercial high-performance parallel computing platforms based on many-core architectures include [2][3][4][5][6].

These architectures are likely to be used, potentially remotely, in systems shared by mutually untrusted parties. For instance, in public and private clouds, where Virtual Machines (VMs) are supplied to untrusted parties on remote platforms, each executing applications coming from different sources, with a different level of sensitivity and trust. These are particularly suitable in a High Performance Computing (HPC) environment.

Given the potential wide use of many-core architectures in security-critical systems and the massive resource sharing enabled by these technologies, it is clear that the issue of security when accessing and handling data on these architectures as well as the protection of personal data for each user are critical.

Many-core architectures must guarantee the integrity and confidentiality of applications as well as the protection of user's personal data in order to ensure their adoption.

While security has been widely addressed in multi-core systems, mainly through logical isolation (see Chapter 2), existing solutions must be evaluated and/or revisited in order to provide a reliable solution suitable for many-core systems.

We particularly focus on software (also called logical) Side-Channel Attacks (SCAs). These attacks allow an unprivileged process to access sensitive information about other processes despite partitioning methods such as memory protection, sandboxing and virtualization.

1.1.1 TSUNAMY project

The national TSUNAMY project (2013-2017) [7] supported by the Agence Nationale de la Recherche (ANR) addresses the problem of secure handling of personal data and privacy in many-core architectures.

The considered system as well as the positioning of the proposed contributions are illustrated in Figure 1.2. In this figure, the hardware and software layers of the system are presented. The hypervisor is responsible for allocating VMs and ensuring the non interference between them. A VM is deployed on a certain number of physical resources on the hardware platform. This latter is heterogeneous and encompasses generic processing ele-

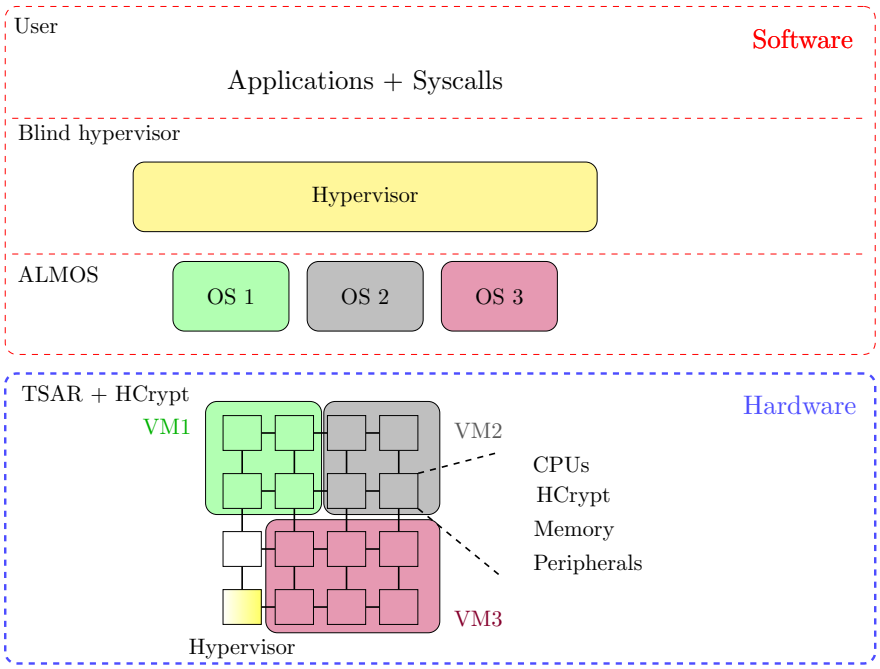


Figure 1.2 – TSUNAMY context

ments for generic computation, as well as dedicated resources for cryptographic algorithms. One VM is allocated to each user. Each VM is considered to run an entire Operating System (OS). Consequently, several OS might run in parallel on different VMs on the hardware platform.

The aim of the TSUNAMY project is then to propose mixed hardware and software solutions allowing to execute numerous independent applications, while providing an isolated execution environment as a response to confidentiality and integrity issues. For this purpose, several partners are involved:

- CEA LIST Commissariat à l'Energie Atomique et aux Energies Alternatives,
- LIP6 Laboratoire d'Informatique de Paris 6, CNRS UMR 7606,
- LabHC Laboratoire Hubert Curien, CNRS UMR 5516 and
- Lab-STICC Laboratoire des Sciences et Techniques de l'Information, de la Communication et de la Connaissance, CNRS UMR 6285.

Several significant contributions are proposed:

- joint development of software layers (driver, API ...) and hardware mechanisms to provide a chain of trust,
- development of a heterogeneous architecture encompassing processing elements to run both, algorithms for processing information and cryptographic algorithms with a strong level of coupling for performance reasons but while ensuring no leakage of information (*TSAR* [8] + *HCrypt* [9] in Figure 1.2),
- development of a software layer with limited rights on the system execution ensuring the physically isolated deployment and execution of VMs (*blind hypervisor* [10] in Figure 1.2),
- development of mechanisms for logical and physical isolation in order to ensure isolated execution of concurrent applications within each Virtual Machine (VM) (*ALMOS* in Figure 1.2) and
- development of strategies for dynamically distributing applications on a many-core architecture (*ALMOS* in Figure 1.2).

The implementation and evaluation of the contributions in the frame of the TSUNAMY project, rely on the TSAR many-core architecture [8] and ALMOS OS [11] (both presented in Chapter 1, Section 1.2.1). The TSAR architecture is enhanced with HCrypt cryptoprocessor [12]. Finally, a blind hypervisor [13] is integrated in order to manage the allocation and management of VMs.

The work of this thesis in the frame of the TSUNAMY project focuses on the secure execution of applications within a VM on the TSAR architecture. The aim is to develop mechanisms and study different dynamic application deployment strategies for the logical and physical isolated execution of concurrent applications.

1.2 Context

In this subsection, the system and the corresponding threat model considered in this thesis work are presented. Finally, logical cache-based side-channel attacks, addressed in this work, are introduced.

1.2.1 Considered system

The TSAR architecture

Many-core architectures are composed of a large number of simple independent processing cores (from tens to hundreds or more). Many-core architectures are divided into clusters composed of processing or memory elements. Clusters can be more complex encompassing several processing and memory elements such as local memory and a cache hierarchy. Within each complex cluster, several cores are connected through a local interconnect and can access some private resources (e.g., private memory, memory caches) as well as some local resources shared among cores within the same cluster as local memory, caches and peripherals among others. Clusters are usually connected through a NoC [14].

Some examples of many-core architectures are the academic TSAR architecture [8], and the industrial Kalray's Massively Parallel Processor Array (MPPA) [2], Mellanox' TILE-Gx36 [3] and TILE-Gx72 [4] processors, Adapteva's Epiphany co-processor [5] and Intel's Xeon Phi [6].

The baseline many-core architecture used in the TSUNAMY project is the TSAR architecture [8], which is a homogeneous many-core architecture with hardware cache coherence and virtual memory support, but no particular mechanism for addressing security issues.

TSAR is a cache-coherent, shared-memory many-core that was jointly designed by BULL, LIP6 and CEA-LETI in the framework of the European CATRENE SHARP project.

TSAR, illustrated in Figure 1.3, is composed of clusters interconnected with a 2D-Mesh NoC. Clusters encompass up to 4 Processing Elements (PEs). Each one has a private level 1 data and level 1 instruction cache (L1). Every L1 has its own Memory Management Unit (MMU) with a separated (instructions and data) Translation Look-aside Buffer (TLB). Additionally, each cluster contains a network interface, some internal peripherals and a level 2 cache (L2) memory bank accessible by all the PEs in the system. In the TSAR version considered in this thesis work, L2 is the Last Level Cache (LLC).

One of the particularities of TSAR is its memory hierarchy. In fact, the memory is logically shared but physically distributed. Every memory location is accessible by every Processing Element (PE) in the system, but the address space (1 TeraByte) is statically partitioned into a fixed number of segments (equal to the number of clusters in the architecture). Each segment is statically mapped on a L2 memory bank (see Figure 1.3). Thus, each cluster L2 memory bank is in charge of one memory segment. Consequently, TSAR

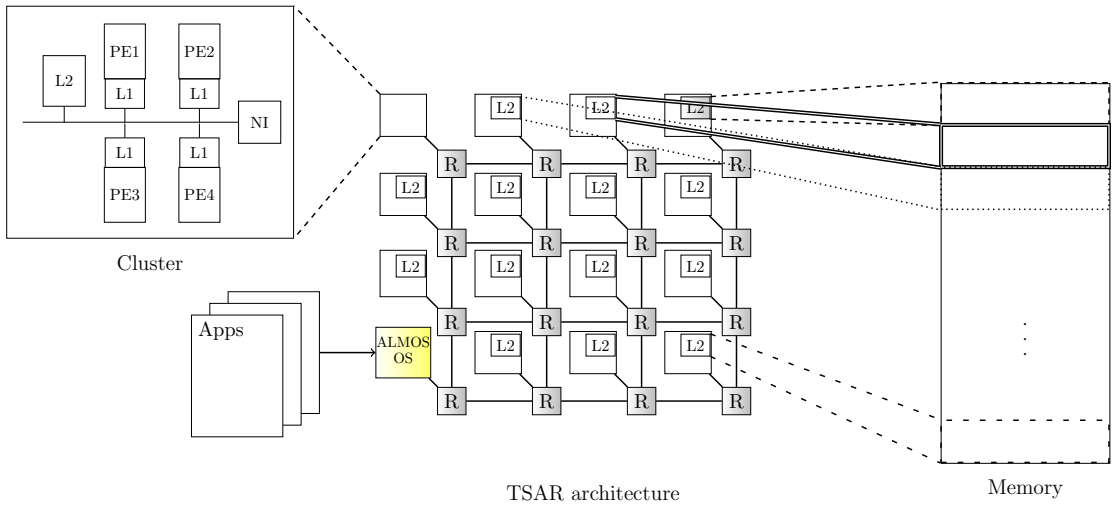


Figure 1.3 – Overview of the considered system

is a Non-Uniform Memory Access (NUMA) architecture. This implies that the memory latency is not uniform but depends on the distance between the PE cluster requesting the memory access and the target memory location cluster.

This memory hierarchy particularity of TSAR, that was initially designed in order to allow a locality aware deployment, is suitable as well for the implementation of the security mechanisms proposed in this work.

Regarding cache coherence, since L1 caches are private and L2 caches are shared, TSAR implements a hardware cache-coherence protocol called Distributed Hybrid Cache Coherence Protocol (DHCCP) [8]. L1 cache implements a write-through policy while the L2 cache implements a write-back policy. Finally, the TSAR architecture is prototyped with a Cycle-Accurate-Bit-Accurate (CABA) SystemC-based simulator [15]. This simulator is able to do accurate full-system simulation at the price of significant simulation time (2000 simulated cycles per second [16]).

The considered system is illustrated in Figure 1.3. It is composed of a TSAR-like clustered NoC-based many-core architecture controlled by ALMOS OS.

The ALMOS OS

ALMOS stands for Advanced Locality Management Operating System [11]. It is an academic UNIX-like OS designed for enforcing the locality of memory accesses made by

parallel tasks applications in order to minimize communication latency and power consumption. The principal OS services required for the fine management of hardware resources are described in Chapter 3, Section 3.4.2.

The TSAR architecture and ALMOS OS are used as the baseline system of this thesis work. The main objective being the protection of the system, in next subsection, the considered attacks are introduced and the associated threat model is presented.

1.2.2 Attacks characterization and threat model

In this subsection, the main guarantees to provide when securing a system are first introduced. After, possible threats and attacks are presented. Then, the Trusting Computed Base (TCB) as well as the threat model associated to the many-core system, considered in this work, are presented.

Secure and reliable systems guarantee the following properties:

- Availability: the availability property guarantees the reliable access to information and to available resources by authorized actors.
- Confidentiality: confidentiality, or privacy, ensures that sensitive information is never reached by unauthorized actors by guaranteeing the respect of a set of rules and access rights.
- Integrity: integrity is the property, that the information is never accessed by an authorized actor. This property guarantees the information is not modified or erased by an unauthorized actor.

However, these properties can be attacked through either, physical and logical attacks. Both categories are presented below.

Categories of attacks

Two different types of attacks are distinguished according to the required access on the system:

- Hardware or physical attacks: these refer to attacks that require direct access to the physical system. In fact, here, adversaries can manipulate the system, can observe and exploit the physical characteristics, and/or modify the physical inputs of the system. These attacks require some specialized equipment and tools in order to measure, modify and exploit the physical characteristics of the device.

Physical attacks can be divided into two main categories according to their level of invasion:

- Non-invasive: Non-invasive attacks interact with the physical device via its physical input and/or output characteristics (voltage, current, clock). These attacks can be passive and only observe and exploit some measurements, or can be active as well by modifying some of the characteristics. However, these attacks do not damage the system and usually do not leave any evidence of the attack on the system.
- Invasive: Invasive attacks on the other hand, require direct access to the inside of the device. Some of these attacks can result in irreversible damage and usually, they leave evidence on the system.

- Software or logical attacks: these attacks do not require any physical access to the system. Here, the main requirement for the attacker is to be able to run his code on the victim's machine.

In the TSUNAMY project, it is considered that the system is remotely used, for example in a cloud environment. Consequently, potential adversaries do not have any physical access to the system and are thus enable to launch physical attacks. Therefore, from now on, we focus on logical attacks only. Based on this scenario, only the software is considered as potentially malicious. The considered threat model is explained in Section 1.2.3.

Threats considered in the TSUNAMY project

Threats considered in the TSUNAMY project are classified according to the threaten security property.

Threats on the availability: Attacks threatening the availability of the system are known as Denial of Service (DoS) attacks. These attacks are intended attempts to stop legitimate users from accessing a specific shared resource preventing them to properly execute. In [17], two main methods to launch DoS attacks are described. The first one consists in sending some malformed data to the victim process in order to confuse a protocol or a running application. A second method aims at disrupting a legitimate user's connectivity by exhausting network or server resources, such as bandwidth, router processing capacity, CPU, memory and disk. For example, one malicious application can request an infinite amount of physical resources in order to endlessly hold the maximum amount of resources.

In this way, other applications cannot use these shared resources and must wait to be able to execute. Under a DoS attack where a malicious process continuously sends traffic to the shared communication infrastructure or services requests, a *non-protected* target system would respond considerably slow or even crash [17]. Finally, concurrent processes, remotely controlled and well organized, can collaborate in order to gain efficiency.

Threats on the integrity: Threats on the integrity concern the unauthorized access by writing data in memory. Malware such as trojan horses and rootkits can try to gain access rights in order to modify data in memory. A malicious process could make a huge number of requests of writing in memory which can cause the overstep on a different process memory region. This can result in unauthorized memory writing.

An attacker can exploit a programming error in a privileged service (e.g, OS services) in order to acquire privileges such as elevated accesses to a normally protected resource or information (privilege escalation) [18]. As a result, the attacker could threaten the availability, confidentiality and/or integrity of the system.

Threats on the confidentiality: Similar to threats on the integrity presented above, threats on the confidentiality target the unauthorized data access, this time by reading it. In the same way as for integrity threats, malware can try to bypass some security policies as well to achieve a privilege escalation.

The MMU and Secure Memory Protected Unit (MPU) are two widely spread countermeasures addressing unauthorized direct access, either by writing or reading, to data in memory. The principle is to verify, at runtime, the respect of access rights for each memory transaction. In this way, processes trying to access some data without having the corresponding right are prevented. If these mechanisms counter unauthorized direct access to data, indirect accesses, through the exploitation of leakage of information, are still possible.

In fact, an attacker process sharing the physical system with a sensitive one, could access and exploit some remanent information after the execution of the victim. Remanent information concerns memory, registers and buffers among others. One solution to avoid remanent information is to clear all the resources used by a sensitive application after its execution.

However, other attacks called SCAs, exploiting a different type of leakage, are more complex and difficult to prevent. These attacks can be physical or logical. As explained above, we focus on logical attacks only. Attacker processes launching logical SCAs observe some information about the victim process such as its execution time and memory access

Mean of threat	Logical attack
Availability	DoS attacks: - Communication bandwidth - Services requests
Integrity	Unauthorized direct access by writing data in memory - Privileges escalation
Confidentiality	Unauthorized direct access by reading data in memory - Privileges escalation Unauthorized indirect access by reading: - Exploitation of remanent data - SCAs

Table 1.1 – Categories of logical attacks according to the threaten security property

patterns in order to deduce some more important information such as secret data or the performed instructions by the victim. These attacks are further explained in Section 1.2.4.

Table 1.1, summarizes the different categories of logical attacks according to the threaten security property.

1.2.3 Threat model considered in this thesis work

The TCB defines the trusted, software and hardware, part of the system on which security policies rely. Therefore, in order to minimize the attack surface, it is important to reduce as much as possible the TCB.

The TSUNAMY project considers that the TSAR architecture is likely to be remotely used, in a cloud environment for instance. It is thus assumed that potential adversaries do not have any physical access to the hardware and therefore cannot compromise it nor launch any physical attack against it. Consequently, all the hardware components are included in the TCB, and only logical attacks are considered.

Applications, external to the system, are always considered as potentially malicious. Moreover, several malicious applications could collaborate in order to attack the system or to attack a concurrent application.

Two scenarios according to the definition of the TCB are distinguished.

The first scenario considers that the entire OS running on the platform is trusted. In fact, in this case it is assumed that the OS kernel services do not include all the features of an entire OS but are restrained to the services necessary for the dynamic deployment of applications and management of resources. In this scenario, only applications running

on the platform are considered as potentially malicious. The OS kernel being trusted, a malicious application can launch attacks against another application but cannot tamper with the OS. Security can thus rely on OS secure-enable mechanisms.

In a second scenario, the OS is not entirely part of the TCB. In fact, here it is considered that, in addition to applications, some OS kernel services can be compromised as well. In this case, it is still required that there is a trusted entity, that can include some OS kernel services or a hypervisor, responsible of ensuring the security policies of the platform.

Finally, concerning both scenarios, there have been some efforts in order to reduce the OS kernel code in the TCB by performing some functionalities, traditionally accomplished inside the kernel, in an outside unprivileged service component. For instance Linux provides a standard User I/O framework for developing user-space-based device drivers. Moving the device drivers into the user space can be done in a security purpose in order to reduce the size of the kernel. Recently, in [19], authors explore this approach.

In this work, the first scenario in which the OS kernel is entirely included in the TCB, is considered. Furthermore, this thesis work considers the execution within one VM and assumes that VMs are securely deployed by the hypervisor which guarantees the non interference between them [13]. Moreover, this work focuses on threats on the confidentiality and integrity and relies on the TSAR architecture supporting an MMU per processing core. The MMU prevents the unauthorized direct access to data in memory by reading and writing. Therefore, among other attacks considered in the TSUNAMY project, this thesis work specially focuses on unauthorized indirect access to data through SCAs. These attacks are further explained in the next section.

1.2.4 Introduction to logical cache-based attacks

SCAs allow an attacker, which has no direct access to critical data, to analyze indirect or side-channel information during or after the execution of a sensitive application (e.g., a cryptographic algorithm) in order to deduce the sensitive application behavior or critical information such as a cryptographic key. Indeed, the implementation of software on the hardware introduces some physical measurements that can be exploited in order to deduce some information about the functioning of the victim's application.

Some examples of side-channels that can be exploited are power consumption [20][21], electromagnetic radiations [22], heat, sound [23] and time variations [24][25]. Depending on the side-channel information to exploit, the attacker may or may not require physical access to the system. We focus on attacks which do not require any physical access (see

Section 1.2.3) but shared physical resources between victim and malicious applications and that exploit time variations. These attacks are called logical or software cache-based SCAs. Especially, we focus on logical attacks that see the cache as the source of leakage. These attacks are not new. In 1998, authors in [26] introduced the prospect of *attacks based on cache hit ratio in large S-box ciphers*. Today, these attacks are still used and improved [24][25][27][28][29][30][31][32][33][34][35].

The cache is indeed a resource that several concurrent processes, sensitive and potentially malicious, compete for. When shared with an adversary, this latter can extract some information about the victim's activity that can be used to perform cryptanalysis.

These attacks can be performed at different granularities. First, these attacks can be performed within a single core when the victim and attacker processes execute on the same core and share the L1 cache [29]. Second, these attacks are also possible across cores when the victim and attacker execute on different cores but share the L2 or L3 cache (i.e., the LLC). Furthermore, these attacks can be performed across VMs in a cloud environment. In fact, a malicious application can perform these attacks against another application on a different VM despite VMs' logical isolation [31][30][32]. These attacks can steal sensitive information from systems implementing logically isolated execution environments [36]. These attacks, originally performed on desktop computers, have recently been extended to NoCs within shared memory Multiprocessor System-on-Chip (MPSoC) [37]. However, these have not been proved on NUMA systems such as the considered system in this work.

Cache-based attacks may be sophisticated, but their underlying idea is relatively simple: an attacker observes cache-based side-channel information such as the victim's execution time or memory accesses in order to gain information about the victim process sensitive data. Additionally, if the attacker can run code on the victim's machine, as well as manipulate the state of the cache, he/she is able to gain some extra information. By exploiting this knowledge, the attacker can retrieve confidential data of the critical program [38].

In state-of-the-art publications, cache-based SCAs are classified as time-driven, trace-driven or access-driven attacks based on the type of information the attacker learns about a victim process [24] [25] [27]. In trace-driven attacks, the attacker learns the outcome of each of the victim's memory accesses in terms of cache hits and misses [39] [28]. Extracting the trace of cache hits and misses in software represents a great difficulty. Consequently, trace-driven attacks are mostly performed in hardware and are thus out of the scope of this thesis work (Section 1.2.3). We thus focus on time-driven and access-driven attacks.

Time-driven attacks

These attacks exploit the vulnerability that, for some algorithms, the execution time is directly related to sensitive data. Moreover, attackers can exploit the fact that the execution time of an application is influenced by the current state of the cache leading to potential leakage of information. There are two categories of time-driven attacks; passive and active. The difference between these two is the location of the attacker. A passive adversary does not have access to the victim's machine and thus, cannot manipulate the state of the cache directly. Here, the attacker process triggers the sensitive application (e.g. an encryption algorithm) a certain number of times and measures the execution time. This latter is influenced by the state of the cache, which is itself influenced by each sensitive application execution. These attacks need more samples than active ones and often require statistical methods in order to successfully retrieve the sensitive information (e.g., the cryptographic secret key). In [40], for instance, a passive time-driven attack is remotely performed on AES algorithm. On the other hand, an active attacker has access and is able to run code on the victim's machine. This allows him to directly manipulate and probe the state of the cache by filling it with its own data or by evicting some specific cache lines. Here, the attacker can trigger the sensitive application, manipulate the state of the cache and measure the execution time. This gives to the attacker additional cache information, compared to passive attacks, and leads to more efficient attacks. A well known technique of this category is the EVICT+TIME presented in [41]. Authors perform an active timing attack on AES showing its efficiency compared to the passive attack presented in [40].

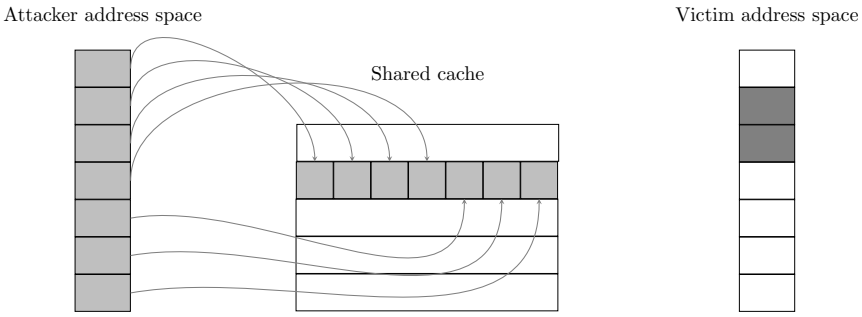
Here, the attacker is able to trigger the AES encryption and to know when it has begun and ended. It is also assumed that the attacker has the knowledge of the virtual memory address of AES lookup tables (T) using input-dependent indices, denoted $V(T)$. Given a chosen plaintext p, one measurement routine for the attacker proceeds as follows:

(a) Trigger AES encryption of the chosen p,

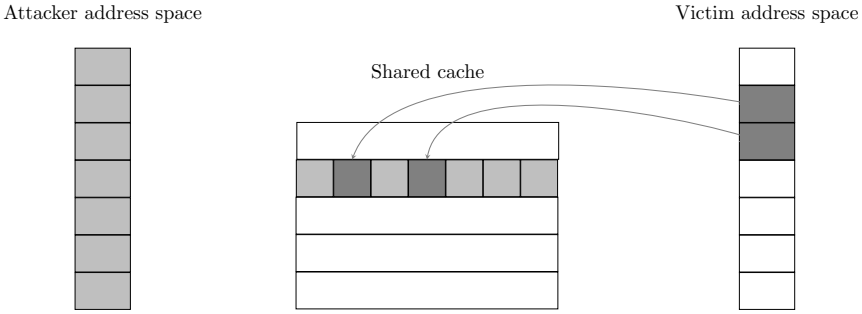
(b) ($EVICT$) Access some memory addresses, B bytes apart, congruent to $V(T)$,

(c) ($TIME$) Trigger a second encryption of p and time it.

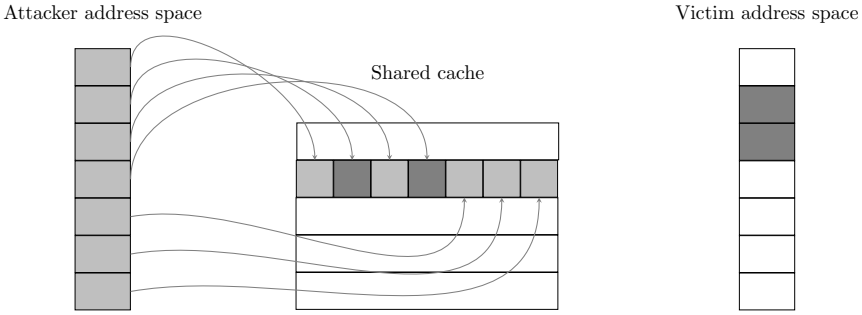
This routine is repeated a certain number of times. The measured time will depend on the plaintext, chosen by the attacker, and on the state of the cache, which is manipulated by the attacker at each routine. By analyzing results, the attacker is able to know which pages eviction influenced the victim's execution time, which will indicate that these pages were accessed by the victim.



(a) PRIME step: The attacker fills the cache with its own data



(b) Wait step: Let the victim access some cache lines during the sensitive computation, consequently, some attacker data will be evicted



(c) PROBE step: Access to data and time. Access to data in cache (light gray lines) will take less time than accessing to data not longer in the cache after being evicted by the victim process (dark gray cache lines)

Figure 1.4 – Principle of PRIME+PROBE [29] attack

Access-driven attacks

In the same way as active time-driven attacks, access-driven attacks rely on the fact the attacker has access to the victim's machine and that there is a shared cache between the attacker and the victim processes. In fact, these attacks exploit the vulnerability that, for some systems, some instructions are related to sensitive data. The principle of these attacks is to deduce which cache lines the victim has accessed by directly manipulating and probing the shared cache and observing the memory access time. This additional information about the victim's cache access patterns makes these attacks more efficient than time-driven attacks.

The PRIME+PROBE [29] is a well known technique. Assume that an attacker manipulates the state of the shared cache by accessing some specific memory addresses, thus, filling the cache with its own data (PRIME). Then, the victim runs for a certain time and potentially changes the state of the cache. Finally, the attacker measures the time to access the same memory addresses again (PROBE). A short access time would indicate that the attacker's data is still in the cache (a cache hit) and thus that the victim has not accessed this cache memory line. On the contrary, a large access time would indicate a cache miss which indicates that the victim has accessed the same cache memory line. By exploiting this technique, the attacker infers information about the memory locations accessed by the victim, and thus the instructions or data that have been accessed.

The attack routine, illustrated in Figure 1.4 proceeds as follows:

(a) (*PRIME*) Fill the cache with its own data,

(b) Wait for the victim to execute and to potentially access some cache lines

(c) (*PROBE*) Access to data and measure the access time in order to determine which cache sets have been accessed by the victim.

Finally, these attacks can be performed both, when the attacker and victim processes execute within the same execution core, as well as when they execute on different cores.

Among the same execution core: Initially, cache-based attacks were performed through L1 caches. In fact, access-driven attacks can be performed in multithreaded system when two processes, an attacker and a victim processes, are concurrently running on the same core and thus share the same L1 cache. In [35], authors demonstrated this technique on a 128-bit AES implementation of OpenSSL 0.9.8n on Linux.

Across-cores: The focus of cache-based attacks has shifted from first-level to shared LLC [30] [42] [33], enabling to perform these attacks across cores. The FLUSH+RELOAD

technique [31] for instance, targets the LLC. Consequently, to launch this attack, the victim and the attacker programs do not need to execute on the same core. This attack extends the technique presented above [35] with adaptations for multi-core environments. Furthermore, the FLUSH+RELOAD attack is a variant of the PRIME+PROBE technique [29] but relies on shared pages between the victim and the attacker in order to monitor the victim access to some specific memory lines. Here, the attacker exploits the inclusiveness property of Intel LLC; every data on lower caches is cached as well in the LLC. Consequently, the attacker can evict a specific cache line (e.g., through a specific assembly instruction such as *CLFLUSH* ([43] in x86) from the LLC which will in return evict the line from all the lower level caches.

A round of attack of the FLUSH+RELOAD [31] technique consists of three phases: In the first phase, a monitored shared memory line is flushed from the cache hierarchy. During the second phase the attacker waits letting some time to the victim to execute and to potentially access the monitored cache line. Finally, in the third phase, the attacker reloads the monitored cache line measuring the time needed to load it. If the victim accessed the cache line during the waiting phase, then the line will be accessible in the cache and the load time measured by the attacker will be short. On the other hand, if the victim did not access the line, then, when reloaded, the line will be fetched from the main memory and the measured access time will be significantly longer. In [31], Yarom et al., present some implementations of this technique. However, attackers might request a significant number of reloads which can be detectable. Thus, a variant of this approach in order to prevent attackers from reloading is to replace the reload phase by a second flush phase (FLUSH+FLUSH) [34].

Across-cores access-driven attacks have been proven practical across VMs [32]. Furthermore, in [37], a PRIME+PROBE-based attack has been implemented on a NoC-based MPSoC.

While cache-based SCAs are often performed against cryptographic algorithms, the techniques presented above are generic and can be used to eavesdrop other non-cryptographic applications in order to recover sensitive (e.g., personal) information. In [34] for instance, authors have used the FLUSH+RELOAD technique in eavesdropping on keystroke timings.

Finally, the attack principle explained in this section exploiting time variation due to threads competition for physical resources, in this case caches, have also been proven practical for other resources such as TLB and Branch Target Buffer (BTB) for exam-

ple [44][45][46].

In this subsection, time-driven and access-driven cache-based SCAs have been presented. In this thesis work we focus on active time-driven and access-driven SCAs, which require a shared cache between the attacker and the victim processes.

1.3 Contributions

In this context, intensive work, including hardware and software mechanisms, has been conducted in order to provide both, resource isolation and countermeasures against cache-based SCAs. However, while hardware solutions require significant changes on the architecture and it might take a while until such changes are available on the market, software solutions, more flexible, often propose application-specific countermeasures or offer probabilistic SCA protection only. Moreover, the proposed approaches have not been designed for many-core architectures and require being redesigned and evaluated in order to be suitable for these new technologies. This thesis work aims at proposing novel contributions taking advantage of many-core characteristics, able to ensure SCA protection on these technologies. For the evaluation of the contributions within this thesis, we consider the TSAR architecture and ALMOS OS as a base system. Towards this objective, the main contributions of this thesis are the following:

- **Spatial isolation of security-critical applications on many-core systems**
  - — Proposal of the spatial isolation of security-critical applications for many-core architectures against cache-based attacks
  - — Proposal of different dynamic deployment strategies for the management of physically isolated execution environment's *secure zones*
  - — ALMOS OS extension in order to integrate the mechanisms responsible for the dynamic management of secure zones on the TSAR architecture

  This work has been presented as a poster presentation at the COMPAS'16 national conference [47], the ICECS'14 [48] and CHES'15 [49] international conferences, as a regular presentation at the PDP'16 [50] and ReCoSoC'16 [51] international conferences, as a presentation at the CrypArchi'15 and CrypArchi'16 international

workshops [52] and as invited talk presentations at the TRUDEVICE'16 international workshop [54] and the DGA-IRISA national security seminar [53] in 2017.

- **Extension of the MPSoCSim virtual prototyping tool**

  — Extension of the MPSoCSim simulator for the exploration of NoC-based multi and many-core architectures design, application deployment and resource management

  — Exploration of the available results and capabilities of the extended version of MPSoCSim

  This contribution has been realized in collaboration with the Application-Specific Multi-core Architectures (MCA) Group at the Ruhr University of Bochum (RUB), Bochum, Germany under the direction of Prof. Diana Goehringer in the context of a 4-month researcher mobility at the RUB University, Bochum, Germany.

  This joint work has been presented as a regular presentation at the ViPES as part of SAMOS XVI international conference [55].

- **Implementation and evaluation of different secure zones deployment strategies**

  — Definition of different performance indicators for the evaluation of the proposed security-enabling mechanisms

  — Implementation and evaluation of different deployment strategies for spatially isolated applications through virtual prototyping

  — Integration of spatial isolation mechanisms within the ALMOS kernel on TSAR within the SoCLib environment

  This contribution has not been published yet but has been submitted to ACM Transactions on Embedded Computing Systems (TECS) as a journal paper and is currently under revision.

- **Dynamic memory-to-cache mapping in the TSAR-ALMOS system**

— Preliminary study on dynamic memory-to-cache mapping in a TSAR-ALMOS context in order to reduce the spatial isolation performance overhead.

This last contribution concerns a preliminary study and has not been published yet.

1.4 Organization of the manuscript

Besides this first chapter, this thesis manuscript is composed of five other chapters:

Chapter 2 first presents main state-of-the-art mechanisms aiming at logically and/or physically isolating resources. Then, it focuses on both, software and hardware countermeasures against cache-based SCAs. Finally, a discussion comparing the presented related work concludes this chapter.

Chapter 3 starts by introducing the principle of the spatial isolation proposed in this thesis work. Then, different proposed deployment strategies for the implementation of the spatial isolation technique are explained. After, this chapter proposes the extension of the ALMOS OS kernel in order to integrate the proposed spatial isolation enable mechanisms. For this, the threat model and the assumptions made for this implementation are first given. Finally, the concerned kernel services in their original state as well as their extension are presented.

Chapter 4 introduces first our motivation to use MPSoCSim for the validation and evaluation of the spatial isolation mechanisms proposed in this thesis work. Then, the original version of this simulator as well as its validation are explained. After, the extensions of this simulator in order to meet our requirements as well as its new capabilities are presented. Finally, results generated by the extended version of MPSoCSim on different architectures and execution scenarios are presented in order to discuss its capabilities.

Chapter 5 presents results generated with the extended version of MPSoCSim in order to compare the spatial isolation strategies proposed in this thesis work. For this, this chapter first presents the experimental protocol used for these experimentations. A case

study showing the cache attacks vulnerability of applications on the execution scenario considered in this work is then presented. After, this chapter presents results comparing the proposed application and resource allocation mechanisms for spatial isolation under several execution scenarios according to different performance indicators.

Chapter 6 concludes this thesis manuscript. First, it summarizes the main contributions presented in previous chapters. Then, it discusses the spatial isolation proposed in this thesis work in terms of related work and possible improvements. Finally, some possible leads for future work, some of these currently studied, are explained.

Chapter 2

State of the art

Chapter contents

2.1 Software and hardware cache-based attacks countermeasures . 23

2.1.1 Software cache-based attacks countermeasures 24

2.1.2 Hardware cache-based attacks countermeasures 27

2.2 Logical and physical isolation 33

2.3 Discussion 40

Many efforts have been done in order to cope with confidentiality and integrity issues. This chapter focuses first on software and hardware countermeasures specifically against logical cache-based SCAs. After, solutions addressing logical and/or physical isolation for confidentiality and integrity are presented. Finally, these state-of-the-art solutions are compared and discussed.

2.1 Software and hardware cache-based attacks countermeasures

In this thesis work, we have particularly focused on micro-architectural timing attacks (i.e., active time-driven and access-driven cache-based SCAs) presented in Section 1.2.4. In this section, software and hardware state-of-the-art countermeasures against the considered SCAs are presented.

2.1.1 Software cache-based attacks countermeasures

Disabling cacheability

A naive solution in order to cope with information leaked by cache utilization is to disable caches. The idea is to modify the implementation of sensitive applications in order to avoid any cache access in order to prevent from cache leakage that might be useful for SCAs. Different implementations of some classic cryptographic algorithms have been proposed. In [38], authors focus on AES algorithm [56] and propose several implementations but arrive to the conclusion that disabling CPU's cache mechanisms will have a devastating effect on performance.

Cache flushing

Another *straightforward* solution would be to flush part of or all the caches after a VM switch in cloud computing for example, as in [57], in order to protect applications from one VM against applications from a different VM. On a single-threaded processor, all caches (local state including TLB, and BTB must be flushed during every context switch. On the other hand, on a processor supporting simultaneous multi-threading, this approach will require the logical processors to use separate logical caches, statically allocated within the physical cache. This solution would prevent a malicious application from knowing which cache lines have been accessed by the victim (access-based SCAs). Moreover, a malicious application that observes the changes on the victim's total execution time influenced by the current data on the cache (time-based attacks) would not detect any change, making these attacks impossible to perform. The main drawback of this solution is the systematic flushing cost itself (authors in [58] benchmarked 8.4μs direct cost for flushing the L1 cache on a 6 core Intel Xeon E5645 processor using clflush instruction) as well as the induced performance overhead as this approach prevents optimal use of the cache. In order to reduce the induced performance overhead, in [59], authors propose to periodically clean the cache to mitigate side channels in time-shared caches providing a less expensive solution with a cost of probabilistic protection.

Partial cache flushing

In the same way, in order to reduce the induced performance overhead of flushing, caches can be only partially flushed [60]. In fact, assembly instructions such as *CLFLUSH*

allow the deactivation of a cache line from the cache hierarchy. This approach can be used by the kernel or the victim process in order to erase some cache parts. These can be the cache parts corresponding to the victim's sensitive data, which would require to know the victim's application security-critical code sections to erase from the cache, or they can be randomly selected in order to obfuscate the attacker measurements.

Disabling high resolution time stamps

This approach relies on the statement that cache-based attacks require to be able to distinguish a cache hit from a cache miss. In order to do so, the attacker requires a time resolution on the order of tens of nanoseconds (considering around 200 cycles difference between a cache hit and miss [31] for a 3GHz Intel i7 core). The aim is then to be able to eliminate high resolution clocks without preventing legacy applications to properly and efficiently execute.

This approach requires to disable high resolution time stamp counters (e.g., ReaD Time Stamp Counter, RDTSC in x86) or to prevent potentially malicious VMs or applications from using them. However, measuring a difference of time can be achieved by other means. In [29], the authors explain that a virtual time stamp counter can be obtained in multi processor systems by using a second thread which repeatedly increments a memory location. Moreover, this is not an easy-to-implement approach, since many applications rely on stamp counters and require them to be available, either for *profiling purposes or to be used in combination with random inputs as source of entropy* as explained in [29]. The same authors propose instead to limit the frequency at which the time stamp counter can be read.

Reducing the precision of the potential attacker VM or application time stamps has been proposed as well. In [61], the authors modify the Xen hypervisor on a x86 platform in order to degrade the resolution of RDTSC and show that their system remains stable under some granularity perturbations. However, in [40], the author claims that this approach does not prevent the attacks but requires from the attacker to average the results over a larger number of samples in order to compensate the induced noise.

Disrupting the attacker measurements

An idea, similar to degrading the precision of time stamps, is to limit the ability of malicious VMs or applications to obtain accurate timing measurements by adding some

noise to the observed timings. This approach does not avoid cache attacks but makes it more difficult for the adversary. In [59], the authors suggest to add some noise between two attacker cache probings in order to obfuscate the attacker measurements. For this, they propose a system for public clouds in which a VM running a guest OS can protect its own execution by adding noise to its caches frequently enough to confound the attacker timings. The noise is added by a kernel thread that is frequently invoked. This latter primes in random order every cache entry until all entries are evicted. Shorter noise adding frequencies entail better statistical protection but greater performance overhead.

Minimum time slices

This approach focuses on PRIME+PROBE attacks in systems without Simultaneous Multi-Threading (SMT) support. In fact, one of the attacker's difficulties is to regain control of the shared CPU resources frequently enough in order to obtain enough prime+probe measurements. The countermeasure presented here is based on this statement and proposes to enforce a minimum time slice for security-critical processes to execute in order to prevent the attacker to obtain measurements sufficiently frequently to be useful. The idea is then to reduce the frequency of victims preemption in order to reduce the interactions with potential attackers. In [62], the authors perform a PRIME+PROBE like attack on ElGamal and show that a minimum CPU preemption frequency of 16μs in average is required in order to be able to perform the attack. In [58], the authors focus on adjusting the hypervisor (but could be the OS as well) core scheduling in order to limit CPU preemptions. However, while focusing on PRIME+PROBE like attacks on systems without SMT support, this solution does not address other access, time-driven attacks, nor SMT systems.

Modifying the implementation or traces of critical processes

A common approach to protect sensitive applications, a cryptographic code for instance, is to make sure that its behavior is never data-dependent. This technique is called *constant-time* [63]. In fact, some cache-based SCAs exploit the victim code cache accesses or branch sequences, as well as overall execution time (see Section 1.2.4). In a cryptographic context, this countermeasure consists at ensuring that cache accesses, branches and execution time of the cryptographic code are independent of the key and the plaintext. In [64], authors give a proof that constant-time programs do not leak confidential information through the cache.

On the other hand, in [40], the significant difficulty involved in ensuring these properties is highlighted. Some efforts have been done in order to guide the design of constant-time code. In [65], a compiler that automatically eliminates control-flow dependencies on secret keys on cryptographic algorithms is presented. Other tools [66], aim at tracing the flow of the secret data in order to detect if it is used in branches or as memory indexes. However the constant-time approach presents some important drawbacks. First, this is an application-specific technique. Second, a constant-time implementation on a given hardware platform may not perform constantly in a different hardware platform. Consequently, the technique must be applied for each application and each hardware platform. Third, constant-time applications usually result on less efficient implementations [38].

Hardware alternative approaches for cache-based SCA protection are presented below.

2.1.2 Hardware cache-based attacks countermeasures

Cache isolation

Page coloring: Page coloring is a well known technique offering cache isolation between memory sections belonging to different processes. The main idea of this approach is to assign colors to memory pages and to ensure that same color pages are mapped to a fixed set of cache lines (Figure 2.1). In [67], the author introduced the concept of partitioned cache as a solution against access-driven cache-base SCAs. In [68], the authors considered page coloring for shared LLC in order to prevent cache sharing. Indeed, this solution prevents a process from influencing or observing the state of cache partitions other than its own partition.

In order to implement cache partitions, the Instruction Set Architecture (ISA) is extended with new instructions able to define a cache partition of a specified size [67]. Page coloring can be implemented both, statically and dynamically. In the static approach, the cache is statically partitioned. A process might not entirely use its partition, entailing a significant number of unused cache lines.

Page locking: To cope with this drawback, the authors in [69] propose to use a similar approach called *Partition-Locked cache* (PLcache). This approach aims at locking in cache only the cache lines of interest (e.g., AES or RSA tables), in order to prevent cache accesses that do not belong to this locked partition from evicting them. Notice that this solution targets both, internal and external cache locked partition interferences. In order to implement this approach, the authors extend every cache line with a tag indicating if

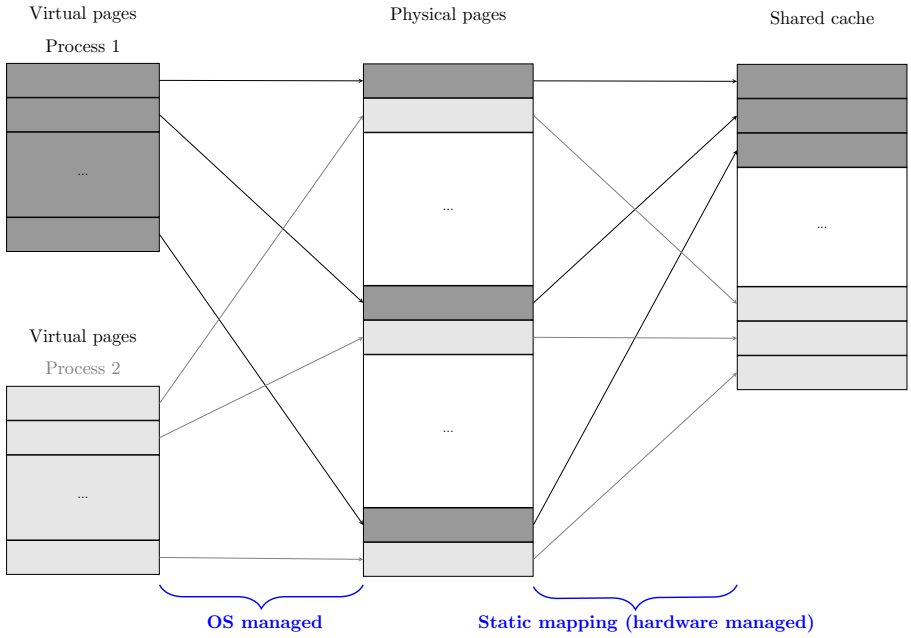


Figure 2.1 – Cache partitions using page coloring

the line is secure and an identifier of the owner of the cache line. The authors propose two methods in order to implement this approach; extending the ISA with new locking, unlocking operations, or letting the OS control the cache lines to be locked and unlocked.

Dynamic page coloring: In [70], the authors introduce *Chamaleon*, a dynamic page coloring mechanism that provides cache isolation only during a security-critical process in a multi-tenancy cloud environment. The aim of this work is to defend a sensitive process running on a VM, from other processes in different VMs. Chamaleon provides an interface to enable an application to notify the hypervisor the entering of a security-critical section. The hypervisor assigns a *secure color* to the process in order to prevent any other process running on a different VM on the same hardware to use the same color. When an application requests a secured partition, the hypervisor assigns a secure color to a given partition that will be allocated to this application. If the allocated partition is already used by another process, Chameleon does a page recoloring by swapping all pages currently using the assigned secure color with pages of other colors. Chamaleon mechanisms are implemented in Xen hypervisor [71]. However, this approach is vulnerable to DoS attacks since any application can request one or several secure partitions.

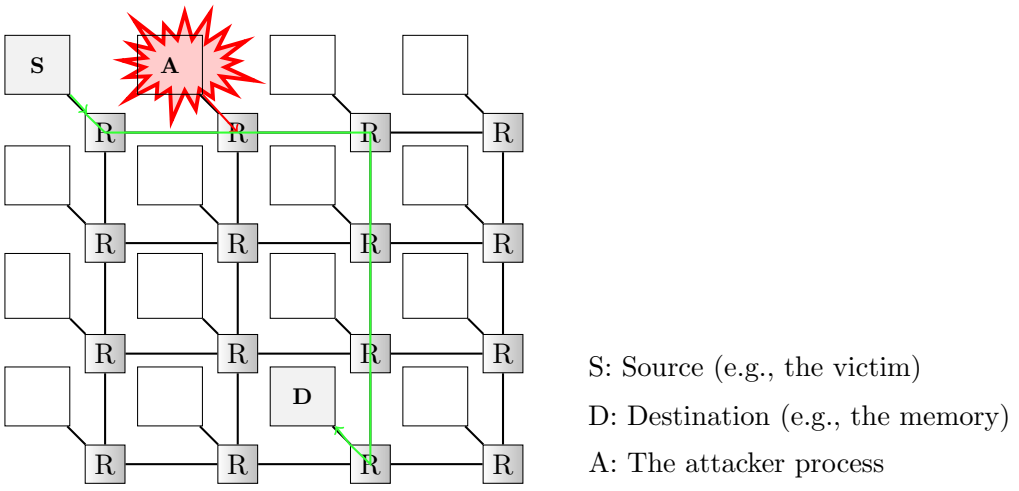


Figure 2.2 – Attack on NoC communication

Disrupting the attacker measurements

Instead of preventing cache sharing, some solutions propose to disrupt the attacker measurements.

Adding noise: Adding noise in order to disrupt the attacker observations is an old technique against timing SCAs. This idea was already explored in [72] where noise is injected into all events visible to a process.

More recently, adding noise has been explored in NoC-based systems [73]. Here the principle is that, for some shared-memory NoC-based systems, an attacker process executing on a node A, located on the communication path between the victim process (S in Figure 2.2), and the shared memory (node D), can observe its own memory access time (or NoC bandwidth) and deduce when the victim is accessing to the memory. This is illustrated in Figure 2.2.

Here, the shared NoC is the source of leakage, but the side-channel can be extended when a cache (L2 or L3) is shared between cores located in different nodes. In a NoC, data transiting is divided into packets. Each packet is routed by every router on the path from the source node S to the destination node D. In order to mitigate the observations of the attacker, the authors in [73] propose to add a random delay in each packet commutation, i.e., each time the packet crosses a router. Packets are then randomly delayed, even if there is no contention and if there is no other concurrent packet competing for the NoC resources.

Consequently, the attacker observations will be not accurate. On the other hand, in order to be able to disrupt the attacker, packets in contented and non-contented scenarios must be confound. Added delay must then be significant enough in order to achieve this.

However, noise injection is inefficient for obtaining high security since it does not provide any strong security guaranty and can significantly degrade system performance [74].

Adding randomness in NoC: In a similar way, the authors in [73] proposed as well to add randomness in NoC router arbiters in order to randomly select the order on which packets are served and routed in a contended scenario. Consequently, the attacker timing measurements will not always indicate a content scenario in which the victim is competing for communication resources but will be obfuscated by the random selection of the packets to route. This technique is proposed to be used in conjunction with other NoC-based approaches such as the previous one suggesting to add random delays in a contention-free scenario.

Furthermore, adopting a non-deterministic routing protocol introduces as well random behavior that can be used to mitigate attacker observations. In [73], using the semi-adaptive west-first routing logic is proposed in order to diminish determinism in the system communication. Fully adaptive routing policies could be used as well.

Randomizing cache behavior: Another solution to disrupt the attacker observations is to randomize the cache behavior. In contrast to cache isolation in last subsection (Section 2.1.2), this approach does not avoid cache sharing, but aims at randomizing the resulting interference, so the attacker observations are not exploitable.

In [69], the authors present the concept of Random Permutation Cache (RPcache). This solution relies on the fact that cache-based attacks exploit the observable internal (i.e., cache interference due to the victim cache utilization) and external (i.e. cache interference due to the attacker cache utilization) cache interferences. It aims then at randomizing both interferences. To do so, this solution proposes to change the miss procedure by detecting interferences between the victim and the potential attacker and by randomizing it through the dynamic memory-to-cache mapping permutation.

Achieving memory-to-cache mapping permutation: Permutations are used to disrupt the attacker observations. Conceptually, the permutation can be implemented by using a level of indirection when indexing the cache. The idea is to store the memory-to-cache mapping for each process in a *permutation table* (Figure 2.3). This latter has the same number of entries as the number of cache sets and each entry contains an M-bit number which indicates a new set. For each cache access, the permutation table is indexed with

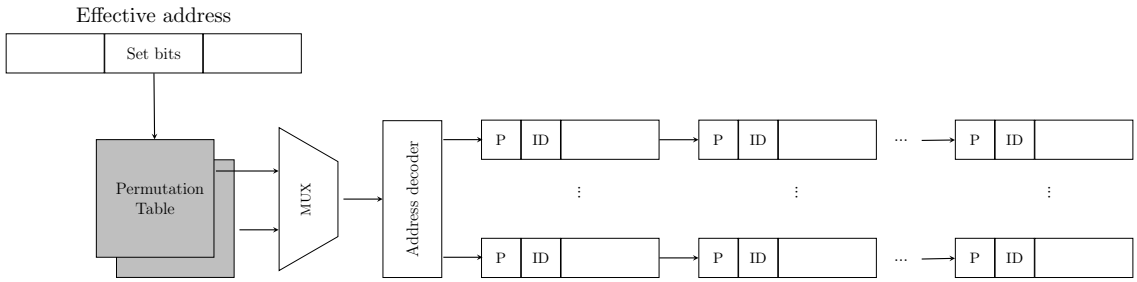


Figure 2.3 – A logical view of the RPcache permutation [69]

the M bits of the effective address (that would normally index the cache set array). The index entry indicates the new M bits required to index the actual cache set array. In order to protect a sensitive process, the knowledge of the attacker process are disrupted by having different permutation tables, for the victim and for the other processes. Each process could have its own permutation table as well.

Cache access handling using permutation: Permutation mechanisms are active each time there is a miss on the shared cache between the victim and the attacker processes. In this case, the replacement policy chooses a cache line R in a set S to be replaced by the requested new data. Last Recently Used (LRU) replacement policy is considered. The proposed mechanism sees if the chosen line to be replaced (R) belongs to the same process requesting the memory access. If not (this case will entail external interference), a new cache set S' is randomly selected where the line R will be replaced. The mappings of sets S and S' are then permuted and the corresponding permutation tables are updated. Otherwise, if the cache line chosen R belongs to the process requesting for the memory access (internal interference), then, a normal miss replacement procedure is done if R has already been permuted. Otherwise, the load/store operation is performed without replacing any cache line. Finally, R is pushed at the end of the LRU queue and this latter is updated in order to choose a different cache line for next replacement procedure. This cache access handling is summarized in Figure 2.4.

In [69], the authors propose a Random Permutation Cache (*RPcache*), a set-associative cache integrating the mechanisms presented above. Its architecture consists in two parts. First, each cache line has been extended with the addition of two tags enabling to know if its mapping has been permuted, as well as the owner identifier of the cache line (P and ID respectively in Figure 2.3). Second, the decoder circuitry has been enhanced in order to implement the indirect indexing illustrated in Figure 2.3 within the L1 cache using the

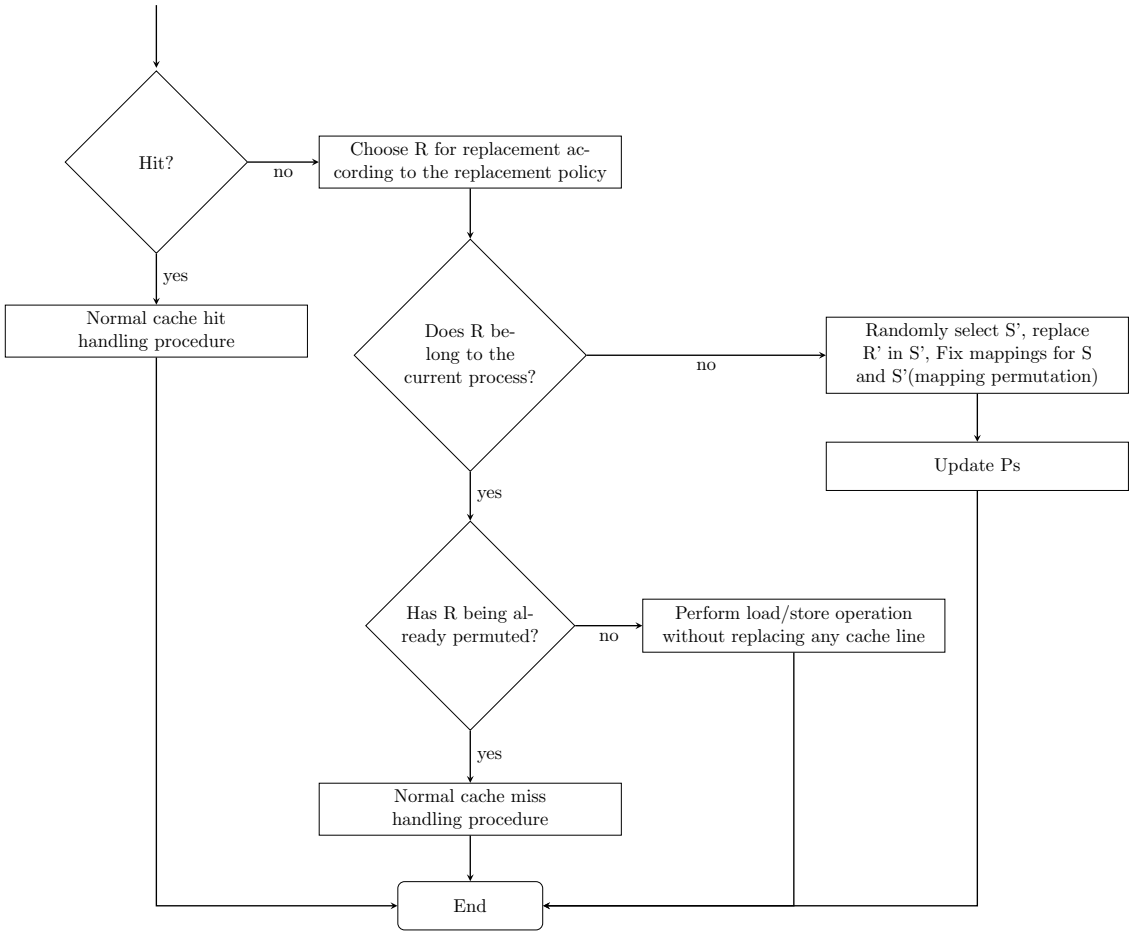


Figure 2.4 – Cache access handling in RPcache [69]

M-Sim v2.0 [75] simulator. The presented results show a performance degradation of less that 2% for the considered execution scenarios.

Another approach aiming at randomizing the cache behavior is presented in [76]. The authors propose an original approach based on a random cache fill strategy. The aim is to dynamically achieve de-correlation between the cache fill and the memory accesses. At each cache miss, the data is sent to the demanding processor without filling the cache. To still benefit from the cache, this latter is filled with fetches randomly selected within a reconfigurable size neighborhood window of the missing memory line instead. Random fetching within a spatial locality of the requested memory, being similar to prefetching, does not entail any performance degradation according to the results in [76]. From the

security point of view, the filling of the cache is not correlated with the memory accesses, which mitigates the information the attacker process can exploit.

Disrupting the attacker measurements by adding noise or random behavior in the system is a well explored approach. However, these solutions, in contrast to strong resource isolation, provide probabilistic protection only and security results could be different according to the sensitive application and execution scenario. From this, it can be concluded that when applications are security-critical, only two different kind of solutions are suitable for guaranteeing a secure application execution with no leakage of information. The first one is considering each application and making sure its implementation will not leak any information (see time-constant implementations explained above). The second one is preventing resource sharing with any security-critical application (i.e., cache isolation). Therefore, in the next section we focus on current approaches for achieving logical or physical isolation as this is a non application-specific approach.

2.2 Logical and physical isolation

Logical isolation

With multiple applications running concurrently and competing for the same physical resources, contention and security issues have been introduced. Processes, which used to manipulate the physical addresses directly, have no more access but to virtual addresses. Virtual memory is an abstraction provided by the OS memory management that offers logical isolation for protection and better memory management. Each process is provided with a virtual address space that differs from the actual physical memory mapping. The management and translation of virtual addresses onto physical addresses is performed by the Memory Management Unit (MMU). The MMU is a hardware component responsible for handling the memory management including memory and caching operations, associated with a processor. The MMU is usually integrated into the processor. However, in some systems it occupies a separate integrated circuit chip. The MMU is in charge of mapping the virtual address space seen by the process running on the processor onto the physical address space existing outside of the processor. The address translation is managed using a translation table, which details, for every virtual address its corresponding physical address, and some other attributes about the memory access such as cacheability and access permissions. The MPU is the component of the MMU responsible for the verification of the rights of every address access. Since each process has its own virtual address

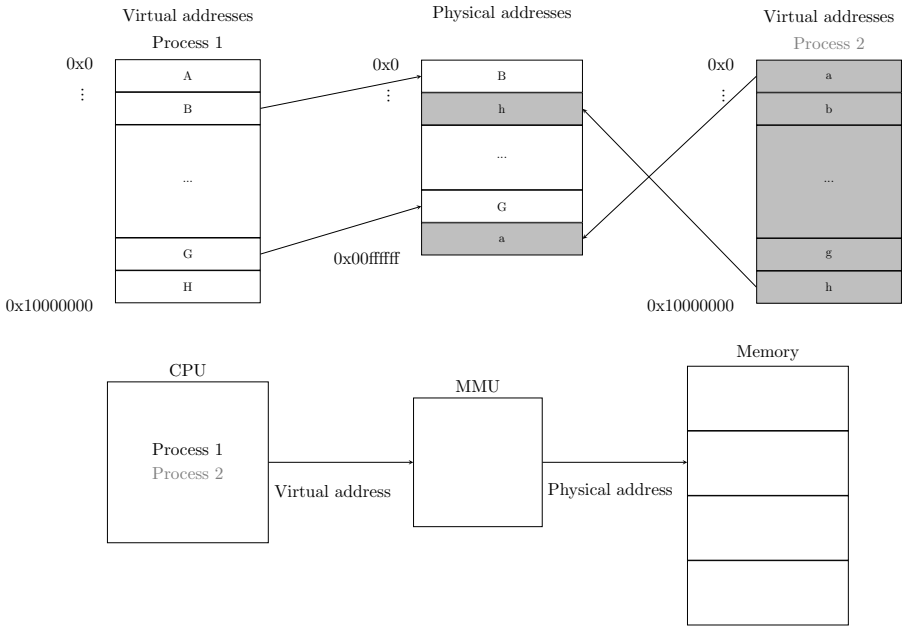


Figure 2.5 – Memory Management Unit (MMU)

space, they are not aware of other virtual address spaces nor physical addresses, and every memory access is mediated by the MPU. These mechanisms ensure that, while physical resources are shared (memory, processors, communication infrastructure), a process cannot (directly) access data in memory without the corresponding right. This guarantee is called logical isolation. The principle of these mechanisms is illustrated in Figure 2.5.

However, while logical isolation prevents from illegal direct access to data, indirect access to data via logical SCAs remains possible (Section 1.2.4).

Physical isolation for security critical data storage and computation

In contrast with logical isolation, some existing hardware components are also able to protect the storage of some specific data within a physically isolated location. Intel Trusted Platform Module (TPM) [77] for instance, provides secured storage for sensitive data, such as security keys and passwords. This module includes as well some encryption and hash functions. TPM is usually used on the boot process in order to ensure secure storage and data integrity before releasing the system control to the OS. This module however is not well bound to the processor which entails slow bus communications. Furthermore, TPM is meant to run only security-oriented code which is supposed to be well written and tested.

Therefore, its utilization is limited to the storage and some authentication primitives for trusted security-critical data.

Trusted Execution Environments and spatial isolation

In order to isolate applications and enforce resource access rights, some existing multi-core architectures provide hardware-based support for the creation of Trusted Execution Environments (TEEs). Examples of these technologies are Intel Trusted Execution Technology (TXT) [78], Intel Software Guard Extension SGX [79], AMD SVM [80] and ARM TrustZone [81].

Intel TXT for example, provides a set of capabilities in order to guard sensitive data from other operations occurring on the same system, including a sealed portion of storage where sensitive data such as encryption keys can be kept and attestation mechanisms in order to make sure the protected code is executing indeed in the protected environment, as well as to verify the system integrity at boot. These capabilities are to be used in order to build trusted execution environments according to the developer requirements.

TrustZone [81], is another example of these technologies. TrustZone is a hardware-based security technology built into ARM CPUs to provide a TEE by switching the entire platform between two different states. Each physical processor provides two virtual cores, one considered non-secure (*Normal or rich world*) and a second one considered secure (*Secure world*), as well as a mechanism to perform context switch between the two modes, known as *monitor mode* [81]. A bit (*NS bit*) in the Secure Configuration Register (SCR) indicates the identity of the current core. The NS bit value is sent on the main system bus to distinguish the virtual core performing an instruction or data access. The non-secure virtual core can only access non-secure system resources, while the secure virtual core can see all the resources.

The two virtual processor cores execute in a time-sliced manner so that only one state is active at a time. The *monitor mode*, in the secure world, is responsible for the context switching when changing the currently running virtual core. The monitor is a security-critical component as it provides an interface between the two states. Figure 2.6 shows an overview of the modes in an ARM TrustZone core.

Within a TrustZone processor the hardware provides two virtual MMUs, one for each virtual processor. This enables each world to have its own set of translation tables. Consequently, each world has an independent control over their virtual-to-physical-address mapping. Each world executes its own OS (rich or secure OS). Finally, the ARM architec-

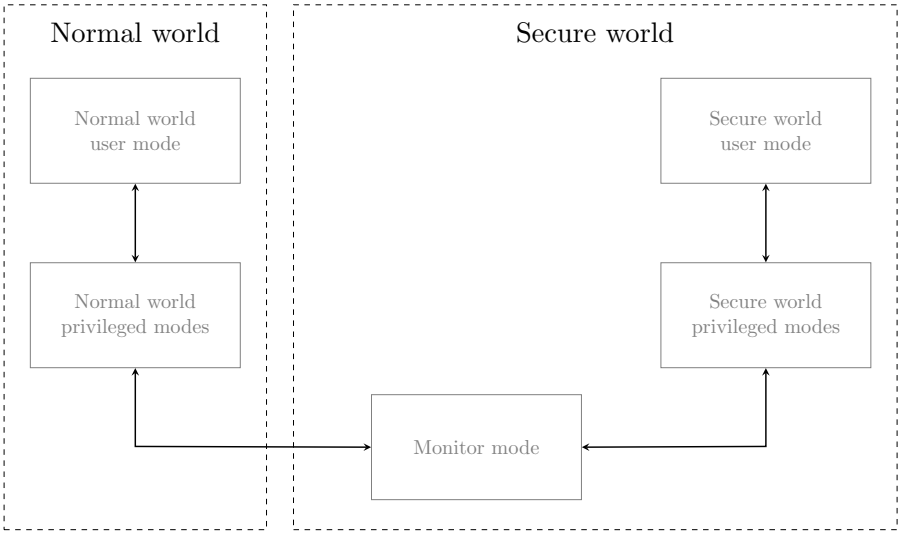


Figure 2.6 – Modes in ARM core with TrustZone extensions [81]

ture includes support for multiprocessor designs with between one and four processors in a cluster [81].

Besides the powerful technology provided by TrustZone, some drawbacks for its use in our context can be highlighted. Indeed, this technology enables two states with different privileges and trust level to run in a time-sliced fashion. As a consequence, this allows only one single trusted compartment within which all sensitive applications execute with no protection against each other. Isolation is then provided at the granularity of the virtual processor. Moreover, very large and complex systems such as many-core are normally composed of simpler small processors (e.g., MIPS) instead of full processors like ARM including TrustZone extensions. Furthermore, since malicious code can share the same CPU as the TEE there is a risk of side-channel communication attacks. Recently, the authors in [82] have shown, for the first time, that the secure world leaks information to the non-secured world. In this work, the authors perform PRIME+PROBE attacks from the normal world on trusted applications (*trustlets*) executing in the secure world during signature verification. The authors show that they are able to observe differences in cache sets according to the validity of keys.

In literature, the authors in [36], address the problem of secure execution of applications in the presence of untrusted system software (i.e., OS, hypervisor, etc). The authors propose to dynamically manage isolated memory compartments (at the memory-page level)

for runtime allocation to critical fragments of code and associated data. For the implementation of this approach, they propose Iso-X, a hardware supported framework including several new instructions that have been added to the ISA and allow within an application to request and manage a secure compartment. Secure compartments are not accessible to any other application nor the OS.

In [83], the authors consider *Security domains* in symmetric multiprocessors for protection of applications. Security domains (Figure 2.7), each composed of one or more processors as well as one OS, are dynamically deployed. A *Base domain* is always deployed and cannot be released. This domain executes the pre-installed applications, considered trusted, as well as a *Context manager* component, a software component responsible for the switching and restoring of CPU contexts when moved to a different domain. On the other side, depending on the number of processors on the architecture, other domains can exist. Indeed, each processor can be added to one or the other domain. For example in Figure 2.7 in step (1), CPU 3 moves from the Base domain to a new domain *Domain A*. Finally, every security domain runs as well context managers that communicate and synchronize for the required context switching. Finally, a *Bus filter* grants access to the bus according to legal resources access for each processor. This approach considers one OS per domain context, each domain isolated from the others. However, applications executing on each domain are not protected against each other. Finally, for the implementation, the authors have implemented a *unified virtual address mapping* for state transition between security domains. This implies that part of the same virtual addresses of every security domain are mapped to the same physical addresses in RAM, which makes easier the switching and restoring of security domains but might at the same time introduce security vulnerabilities since domains are not longer completely logically isolated.

Finally, these technologies consider a single multi-core system, do not consider isolation on many-core architectures and do not explicitly address cache-based SCA.

Resource isolation in many-core systems

In [84], the questions of how isolation can be achieved in current industrial many-core systems and what security properties these architectures can enable, have been addressed. The authors specifically focus on design choices in the construction of many-core systems in order to allow secure partitioning of system resources between applications or VMs in a cloud-like scenario. After a comparison of the security properties that could be enabled by three many-core systems in the market; Tilera [3][4], Epiphany co-processor [5] and

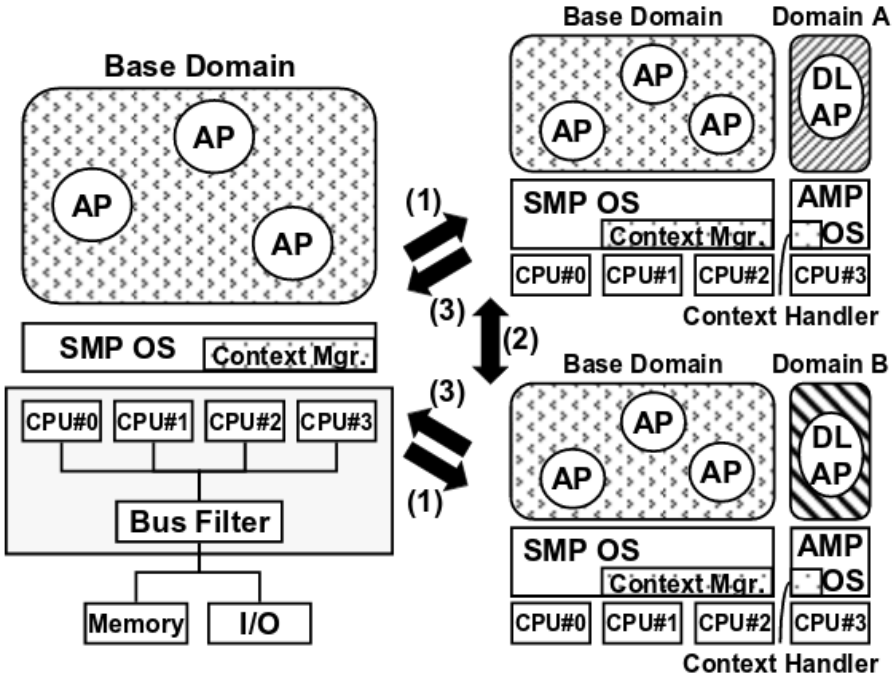


Figure 2.7 – Dynamic security domains [83]

Intel's Single Chip Cloud Computer (SCC) [85], the authors decided to focus on this last one as its design leads to security vulnerabilities in terms of unauthorized access to resources. Their approach is then to implement resource partitioning mechanisms in order to enforce resource access rights. Intel's SCC, is a co-processor suitable for cloud computing applications that consists of 48 x86 cores organized in 24 tiles. Each core has its own MMU which translates virtual addresses to physical addresses. Physical addresses are then translated into *System-wide* addresses using a Look-Up-Table (LUT) at the network interface. Each core has its own LUT and is able to read it and modify it at run-time. The particularity of this platform is that every core can access any system resource, except from caches, by modifying its own LUT leading to vulnerabilities of confidentiality, integrity and DoS attacks. Therefore, the authors implemented *SEMA*, an SCC extension, in which the configuration of the LUTs is made only by the trusted kernel. This latter, considered as part of the TCB, is then responsible for resource access rights and isolation between applications. The main required hardware changes (shown in dark gray in Figure 2.8) are the following: the kernel is the only one that can modify the Look-Up-Tables (LUTs), a *Context aggregator* component is responsible for collecting the status of LUTs (monitoring)

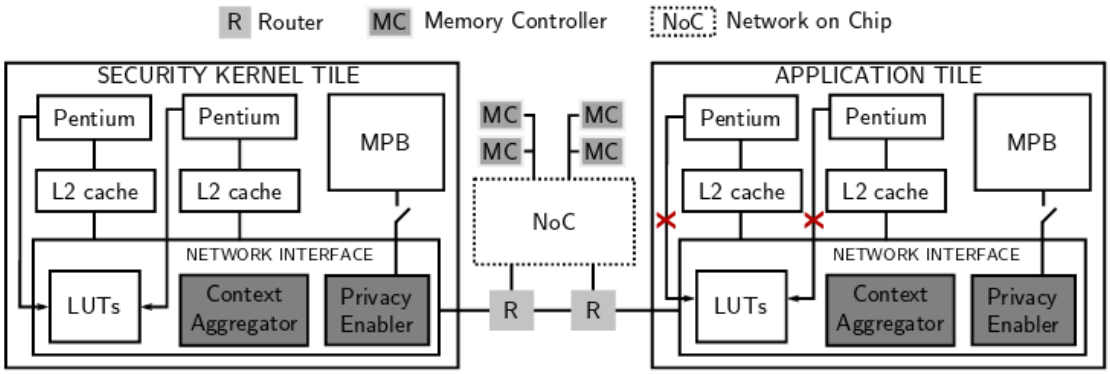


Figure 2.8 – Extension of SCC for resource isolation [84]

and a *Privacy enabler* prevents access to on-tile resources from other tiles.

Finally, the authors introduce as well some new not yet implemented ideas of new security properties such as application context awareness. This property would enable an application to be aware of the context within its execution in order to be able to detect a potentially malicious kernel and protect itself against it. The authors propose as well the requirement of a trusted kernel with restricted rights over applications execution.

While this work enforces resource access rights, it does not addresses cache-based attacks, but focuses on illegal direct access to resources. Moreover, the implementation of the proposed approach is specific to the SCC architecture.

Theoretical scheduling analysis for spatial isolation in MPSoCs

In [86], similarly to us, the authors consider the spatial isolation of sensitive applications. The authors focus on tile-based NoC MPSoCs and propose a theoretical scheduling study in order to optimize the resource allocation. The study considers a scenario in which the applications to execute are known and proposes a *hybrid* application mapping approach including off-line application and architecture profiling as well as run-time allocation decisions. Each security-critical application is executed within an isolated partition of the system called *shape* within which all communication and computation is assumed to be performed. Communication through the NoC as well as access to off-chip resources remain to be considered. This approach has been evaluated through theoretical experimentations, but the authors propose to explore this approach on a real system in future work.

2.3 Discussion

In this thesis we focus on the protection against cache-based SCAs on many-core architectures. In the last section, state-of-the-art related work is presented. This latter can be divided into two categories.

The first category concerns solutions against cache-based SCAs. Three different categories, depending on the countermeasure main objective, can be highlighted. The first category aims at avoiding cache sharing. This kind of approach can be achieved through hardware mechanisms modifying the cache architecture and its utilization, such as cache coloring and page locking, or through software, by explicitly changing the security-critical application implementation in order to modify its memory accesses. These approaches are able to guarantee a *strong* protection against cache-based attacks in return to performance overhead and flexibility. On the other hand, the second category of this approach accepts cache sharing but aims at randomizing the induced interference that can be exploited by the attacker. This is mostly achieved by adding random noise when accessing data, or by introducing some random behavior on the cache. These solutions are not generic and require hardware changes that can be significant, specially when changing the cache design. Moreover, in contrast to strong protection provided by first category solutions presented above, these approaches guarantee probabilistic protection only. Finally, a third category is related to constant-time implementations of security-critical applications (see Section 2.1.1). This application-specific approach ensures the strong protection of sensitive applications against cache-based attacks at the price of less efficient implementations.

It can be noticed then that in order to provide strong protection to security-critical applications, in contrast to probabilistic protection approaches, only two kind of solutions are suitable. First, making sure for every security-critical application that its implementation does not leak any information (constant-time implementations), but this is application-specific, or second, preventing resources sharing with security-critical applications (cache isolation).

Therefore, the second category focuses on mechanisms aiming at achieving logical and/or physical isolation of resources. These solutions include mechanisms for the protection of memory access rights, the physical isolation of memory for the protection of security-critical data, as well as the temporal physical isolation of resources such as TEEs and the bi-partitioning of the platform. Existing multi-core platforms already include these solutions. These, often enhanced with hardware mechanisms, avoid some specific security-

	Aim of the countermeasure	Approach category	Confidentiality and integrity guarantee	Cache SCAs protection	SW	HW	Application-specific	Probabilistic protection	Strong protection
Logical and/or physical isolation	Logical isolation for enforcement of memory access rights	Per core MMU/MPU	✓		✓	✓			
		In many-core [84]	✓		✓	✓			✓
	Memory physical isolation	TPM [77]	✓		✓	✓			✓
	Temporal physical isolation of resources	Trusted execution environments [78] [80] [79]	✓		✓	✓			✓
		Platform partitioning [81]	✓			✓			✓
Countermeasures against cache-based SCAs	Avoiding cache sharing	Cache partitioning: - coloring [70]		✓		✓			✓
		- pages locking [69]		✓	✓	✓	✓		✓
		Avoiding cache utilization for sensitive applications by changing its implementation		✓	✓		✓		✓
	Accepting cache sharing but disrupting the attacker observations	Adding random delays [72] [73]	✓	✓	✓	✓		✓	
		Disabling or reducing time stamp accuracy [61]		✓	✓			✓	
		Minimum scheduling time slices [58]		✓	✓			✓	
		Preventing determinism in NoC communication [73]		✓		✓		✓	
		Random memory-to-cache remapping [69]		✓		✓		✓	
		Random cache filling [76]		✓		✓		✓	
	Guaranteeing no leaking of the application	Constant-time implementation [63] [64]		✓	✓		✓		✓

Table 2.1 – Comparison of state-of-the-art countermeasures

critical resource sharing, memory, communication links, peripherals, in order to prevent illegal direct access to resources. However, these solutions do not address illegal indirect access to data through side-channel attacks. Therefore, additional or different mechanisms are required.

Table 2.1 summarizes and classifies the state-of-the-art countermeasures presented in this chapter.

Our work, specially addressing cache-based SCAs, is more related and suitable to be compared with solutions within the second highlighted approach. Many efforts have been done. A significant part focuses on hardware countermeasures. These require changes on the micro-architecture and it might take a while until such a new processor generation is available on the market. Software countermeasures on the other hand, concern application-specific solutions or probabilistic protection only.

In this work, we target protection mechanisms providing strong protection against cache-based attacks. We consider software solutions thanks to their flexibility and reduced cost compared to hardware solutions. In contrast to previous software application-specific solutions presented above, we aim at proposing system level generic methods to be employed at the kernel OS in order to guarantee strong protection against cache SCAs.

Chapter 3

Spatial isolation

Chapter contents

3.1 Spatial isolation 43
3.2 Different deployment strategies for spatial isolation 46
3.2.1 Static size secure zone approach 46
3.2.2 Fully dynamic size secure zone approach 51
3.2.3 Trade-off strategies combining both, static and dynamic approaches 53
3.3 Summary of the proposed deployment strategies 57
3.4 Extension of the kernel services for spatial isolation implementation . . . 58
3.4.1 Threat model and assumptions 58
3.4.2 ALMOS kernel services and integration of spatial isolation 60
3.5 Summary of the extensions of the kernel services 69
3.6 Conclusion 71

In this thesis work, the spatial isolation of sensitive applications against cache-based SCAs on many-core architectures is proposed. In this chapter, this approach is first presented. Then, several deployment strategies for the implementation of this solution are explored. Finally, the extension of the OS kernel services, in order to integrate the mechanisms responsible for the dynamic management of this approach, is presented.

3.1 Spatial isolation

In this work, we focus on active time and access-driven SCAs (Section 1.2.4). The origin of these attacks is cache sharing. In order to thwart cache-based SCAs, we consider

the spatial isolation of sensitive applications. The aim is to avoid cache sharing for every critical application by temporarily dedicating physical resources (*secure zone*) for its execution. Secure zones, are composed of a number of contiguous clusters that are completely dedicated to a single sensitive application during its entire execution time. Secure zone clusters are spatially contiguous in order to minimize the communication latency between tasks within an isolated application. Moreover, secure zones are dynamically deployed, managed and released. In this way, cache sharing with an isolated application is avoided, and thus, cache-based attacks will no longer be possible against this application. Note that several isolated applications can run simultaneously, each one within a separate secure zone (see Figure 3.1). Finally, when an isolated application has been executed, memory within its secure zone is cleared in order to avoid any leakage of information.

Before deploying a sensitive application, a secure zone is created dedicating a certain number of clusters, depending on the chosen deployment strategy (strategies are explained in Section 3.2). The sensitive application executes within the secure zone. Every task created by an isolated task is mapped and executed within the secure zone. Depending on the deployment strategy, secure zone resources might be dynamically added and released from the secure zone. Once the isolated application finishes, its remaining secure zone resources are released and memory within the secure zone is cleared. At this stage, the released resources are declared available again and can be used by other applications.

Figure 3.1 illustrates the principle of this technique for dynamic size secure zones on a clustered architecture. Different consecutive execution times are shown ($t0 < t1 < t2$). At *t0*, there is no load on the platform. Then, at *t1*, three different applications, including *Isolated application 1* requiring to be spatially isolated, are deployed and start their execution. At that time, there are enough available resources. Consequently, applications do not share resources and do not interfere with each other. Later, at *t2*, previous applications have extended on several clusters, secure zone 1 encompasses now two clusters which are dedicated and are not shared with any other application. Further, *Isolated application 2* has been deployed and executes within a 4 cluster secure zone. Other non-isolated applications have been deployed as well. Due to the increasing load on the platform, non-isolated applications are obliged to share their resources with each other and are thus vulnerable to cache-based attacks.

Note that in this approach, the non-isolated applications still use and share caches with other untrusted applications. Indeed, cache sharing is avoided only for sensitive applications.

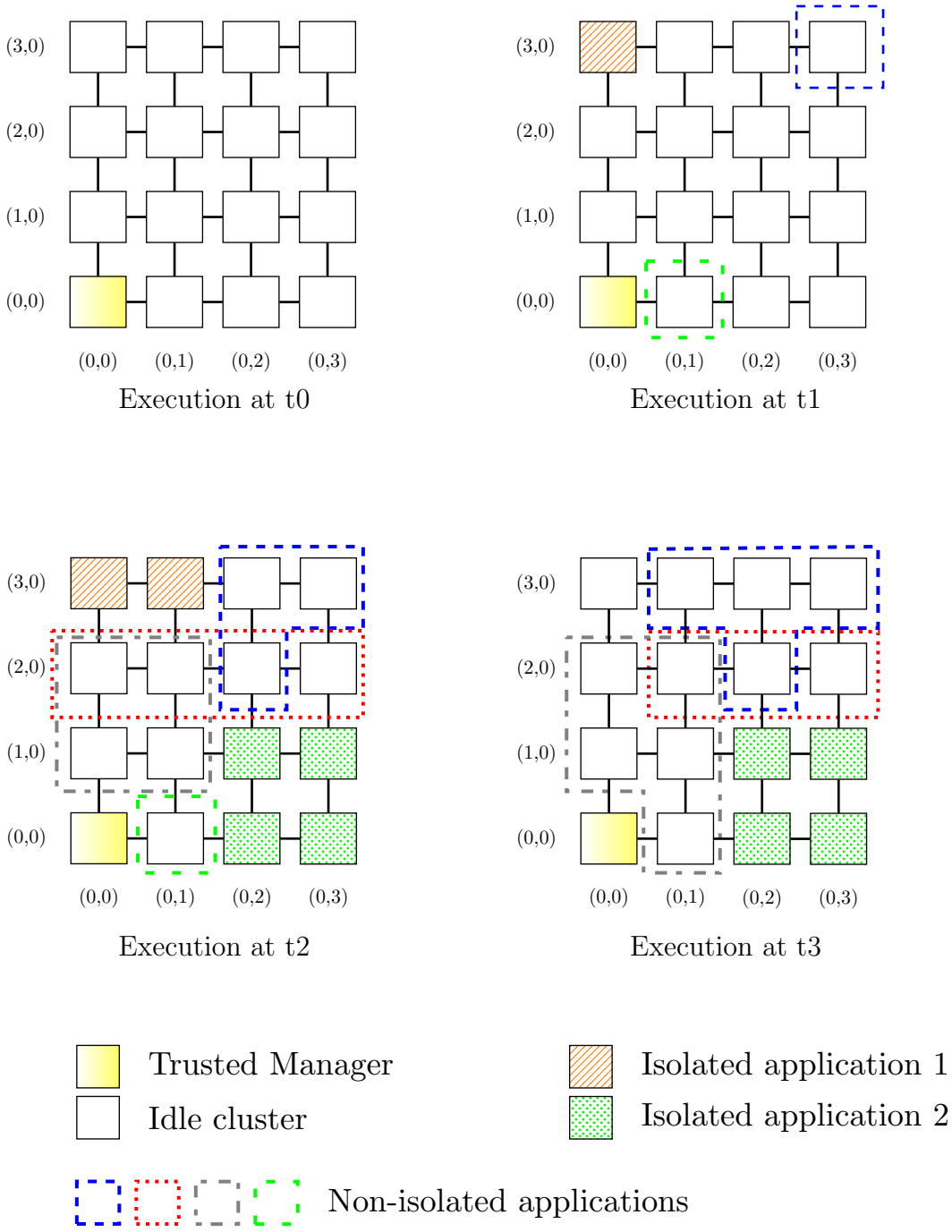


Figure 3.1 – Principle of spatial isolation

We propose to implement the spatial isolation approach by extending the OS kernel services. Additionally to the dynamic management of resources, this latter is responsible as well for the secure-enable mechanisms presented in this work. On the other hand, the dedication of secure zone clusters resources (processing and memory) might introduce an under-utilization of resources and thus, a performance degradation. In order to minimize and handle the induced performance degradation, we propose different deployment strategies for the implementation of this technique and compare them in terms of different performance metrics. The proposed deployment strategies are explained below.

3.2 Different deployment strategies for spatial isolation

Physical isolation entails an under-utilization of resources due to the temporal resources dedication for the execution of security-critical applications within secure zones. This leads to a performance degradation of different performance indicators (see Chapter 5). Consequently, we search at minimizing and managing the performance overhead through different deployment strategies for the dynamic creation, management and release of secure zones. This section explains and compares the proposed strategies.

3.2.1 Static size secure zone approach

A straightforward approach is to create a secure zone of a static size in terms of clusters. Two cases are possible concerning the secure zone size. First, the secure zone is composed of all the resources the isolated application requires in order to achieve its maximum parallelism. Note that this size encompasses computing and memory resources (static memory needs). Second, the secure zone is restrained to a limited size. In this work, it is assumed that the size of the secure zone is known (e.g., previously determined, specified by the user, or through application profiling [87] [88] for example). Before mapping the application meant to be isolated, the secure zone is created by dedicating the number of specified resources. Only then, the isolated application tasks are dynamically mapped and executed within their secure zone.

In the first case presented above in which the secure zone includes all the resources the isolated application needs, each time there is a new task belonging to the isolated application, resources within the secure zone will be available and the task will be mapped within the secure zone without waiting for resources. On the contrary, in the second case, the secure zone does not include all the resources required by the application. Consequently,

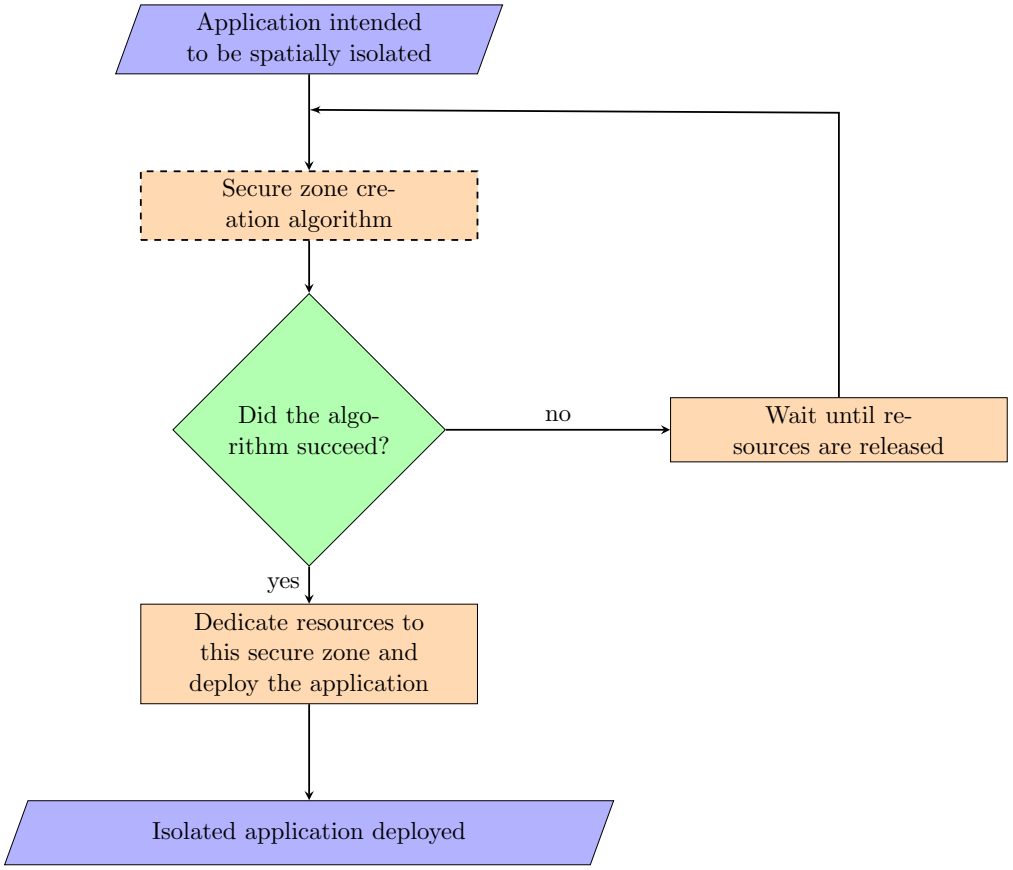


Figure 3.2 – Creating a Secure Zone

some of the tasks might need to wait for available secure zone resources before they can be mapped and executed.

In both cases, once all tasks of the isolated application have been executed, all the secure zone resources are released and are declared available again. Note that if there are not enough available resources to create a secure zone, another attempt will be made when a resource is released. Consequently, the application will wait for available resources to be executed.

Figure 3.2 shows the flow of the secure zone creation for every deployment strategy presented in this section, the dashed block corresponds to a secure zone creation algorithm (see Algorithms 1 and 2). When an application intended to be spatially isolated is scheduled, the kernel runs the algorithm responsible for the creation of a secure zone (aforementioned

and explained in this section). If this algorithm succeeds, then the secure zone is created and all the resources within it are dedicated to the isolated application. Finally, the application is deployed. Otherwise, if the algorithm does not succeed at creating a secure zone, the application will wait for the kernel to try again when resources are available.

Algorithm 1 is responsible for the creation of a fixed size secure zone. This algorithm is the base for every strategy presented below in terms of search for available contiguous clusters. The algorithm receives as input the architecture state as well as the required size of the secure zone (A, and l respectively in notations below). It gives as output the list of clusters if success (here E), and a failure notification if not. For performance reasons, deployment strategies are based on greedy algorithms aiming at finding a solution, in this case l idle contiguous clusters, as fast as possible and not necessarily at finding the global best solution. Figure 3.3 and Algorithm 1 explain the fixed size secure zone algorithm with the following notations (lists are spelled in upper-case letters while single elements and variables in lower-case letters):

— A: architecture,

— P: list of idle clusters in A,

— l: required size in terms of number of clusters for the secure zone,

— E: list of secure zone clusters,

— c: an initial cluster from which the secure zone is created (white cluster in Figure 3.4),

— c': the first cluster in Vc,

— $sort$(list): sort of clusters in a list by the distance between each cluster and the initial cluster c (in ascending order),

— Vc: list of explored clusters from c at current depth d (light gray clusters in Figure 3.4),

— Vn: list of clusters to explore at depth d+1 (dark gray clusters in Figure 3.4) and

— d: current depth of explored clusters from c. Depth is ranged from 1 to l. When d=1, only the initial cluster c is explored (white cluster in Figure 3.4). When d=2, the initial cluster c and its 4 direct neighbor clusters are explored (light gray clusters in Figure 3.4), at d=3, c and its 4 direct neighbor clusters and their neighbor clusters will be explored (dark gray clusters in Figure 3.4), etc.

The algorithm considers each idle cluster as the *initial cluster*, c in Algorithm 1, but stops as soon as it finds a solution. From c, it tries to find enough contiguous clusters by

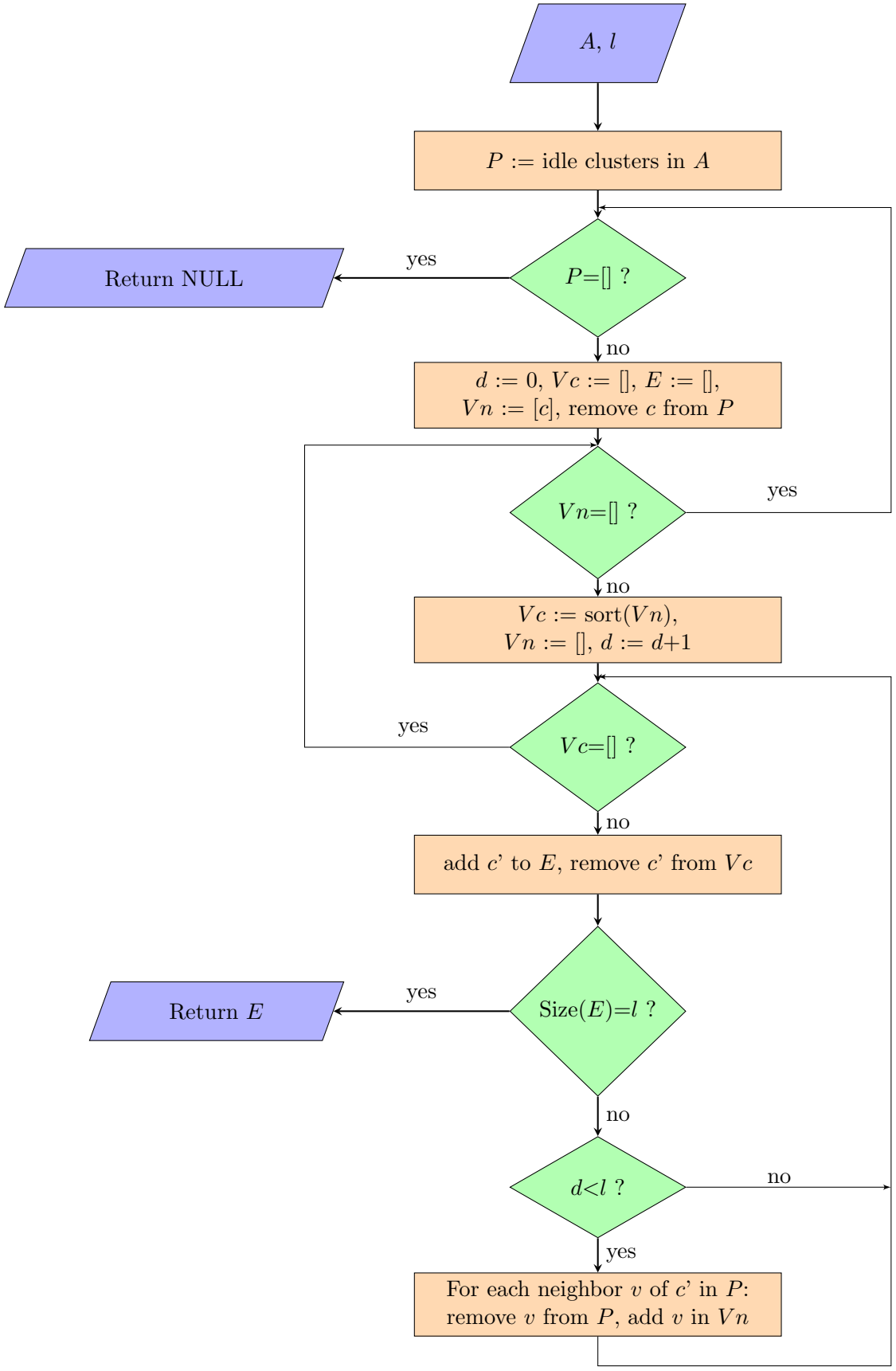


Figure 3.3 – Overview of the creation of a static size secure zone flow

Algorithm 1: Creating a fixed secure zone

```
Input:
l: size of E in terms of number of clusters,
A: architecture
Output:
E: list of clusters in the Secure Zone if success, NULL otherwise
Let be:
P: list of idle clusters in A
Vc: list of clusters from P to explore at depth d
Vn: list of clusters from P to explore at depth d + 1
d: current depth
c, c': clusters in P

while P ≠ [] do
    d := 0
    Vc := []
    E := []
    let c being the first cluster in P
    Vn := [c]
    while Vn ≠ [] do
        Vc := sort(Vn), Vn is sorted by the distance from c in ascending order
        Vn = []
        d = d + 1
        while Vc ≠ [] do
            let c' be the first cluster of Vc
            add c' to E
            remove c' from Vc
            if size(E) = l then
                return E
            foreach v a neighbor cluster of c' in P do
                if d < l then
                    remove v from P add v to Vn
return NULL
```

selecting for each cluster in the secure zone, the idle neighbor cluster the closest to the original cluster c. This, in order to minimize the communication cost between the *father task*, to be mapped on the original cluster c, and the children tasks created by the father task and mapped on the remaining clusters within the secure zone.

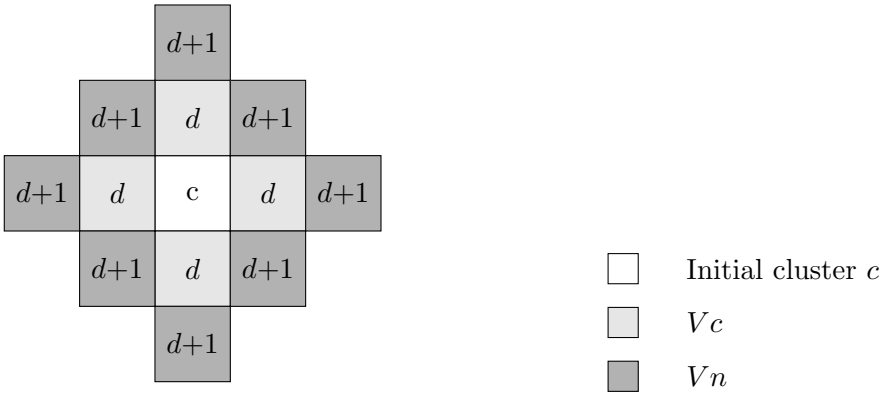


Figure 3.4 – Exploration of clusters in secure zone creation algorithms

— **Advantages:** In this approach, the performance of the isolated application can be favored depending on the fixed size of the secure zone. The application can achieve its maximum parallelism if the secure zone is composed of all the resources needed by the isolated application.

— **Drawbacks:** On the other hand, before being deployed, the isolated application will wait until the necessary contiguous clusters are available in order to create its secure zone. Thus, we can expect a more or less significant wait time before the execution of isolated applications that will depend on the load of the platform and size of its secure zones. Moreover, the resources within the secure zone being dedicated to a single sensitive application during its entire execution time, an under-utilization of resources within the secure zone is expected. Consequently, the untrusted applications are prevented to use the dedicated resources impacting their performance as well as the global performance due to the potential under-utilization of resources within the secure zones.

3.2.2 Fully dynamic size secure zone approach

In this fully dynamic approach, the size of the secure zone is dynamically adapted to the needs of the isolated application according to the load of the platform. For this, the isolated application requires only one single idle cluster to form its initial secure zone (see Algorithm 2), and physical clusters are dynamically added (Algorithm 3) and released from the secure zone. When a new task belonging to an isolated application requires to be mapped, or more memory is needed, then the dynamic approach algorithm first searches

Algorithm 2: Creating a fully dynamic secure zone

Input:
A: architecture
Output:
E: list of clusters in the Secure Zone if success, NULL otherwise
Let be:
c: cluster in A
$E := []$

foreach $c \in A$ **do**
    add c to E if c is idle
    return E
return NULL

for an idle processor (or available memory according to the needs) within the secure zone. If there is no idle processor (or memory) within the secure zone, then it searches for an idle cluster contiguous to clusters within the secure zone. If no contiguous cluster is available, then, there are two possibilities, whether the new task waits until a resource (processor or memory) within the isolated application secure zone is released, or the task waits until a resource on the architecture is released and the kernel adds it to its secure zone. Note that when a non-isolated cluster is declared available (idle) again and an isolated task (or intended to be isolated) is in the pending tasks list waiting to be mapped or when there is a pending memory request from an isolated task, then, the kernel will attempt to extend the corresponding secure zone again.

— **Advantages:** Isolated applications might wait a shorter time than in a static size approach since an isolated application only needs one single cluster to start executing. Moreover, since the size of a secure zone is dynamically adapted, a better utilization of resources is expected. Consequently, the performance of untrusted applications may be less penalized.

— **Drawbacks:** While this approach might entail better resources utilization and less impact on the untrusted applications performance, isolated applications performance will no longer be a priority. Consequently, we can expect that increasing the secure zones size will be more difficult.

Algorithm 3: Extending a dynamic size Secure Zone

Input:
E: list of clusters in the Secure Zone,
A: architecture,
c: original cluster on which the father task is mapped
Output:
E: list of clusters in the Secure Zone if success, NULL otherwise
Let be:
P: list of idle clusters in A
Vc: list of clusters from P
Vn: list of clusters from Vc
c': a cluster in E
c'' a cluster in Vc

foreach $c' \in E$ **do**
  $Vc := []$
  $Vn := []$
  foreach *v a non explored neighbor cluster of c' in P* **do**
    add v to Vn
    foreach if then
    do
      $Vn \neq []$
    $Vc :=$ sort(Vn), Vn is sorted by the distance from c in ascending order
    let c'' be the first cluster of Vc
    add c'' to E
    return E
return NULL

3.2.3 Trade-off strategies combining both, static and dynamic approaches

After studying these two opposite solutions (i.e., fully static and fully dynamic deployment strategies), we considered less extreme strategies combining both, static and dynamic approaches. Two different trade-off solutions are presented in this subsection.

Dynamic approach with a guaranteed non-optimized secure zone size

A variant of the two explained approaches is to dedicate a non-optimized number of clusters (l' specified by the user) to an isolated application before executing it, and to dynamically add resources while needed following the approach presented in Algorithm 3. An overview of this approach is given in Figure 3.5. In this approach, the isolated application needs to wait until the secure kernel finds the minimum number of clusters specified by the user to form a secure zone before starting its execution. The secure zone will be created following Algorithm 1, where the input parameter l is the user specified secure zone size (l becomes l'). Once the secure zone is created and the secure kernel launches the execution of the isolated application, additional clusters can be dynamically added and released, but the original clusters, on which the secure zone was created, will remain dedicated during the entire isolated application execution time.

— **Advantages:** This solution guarantees a minimum size of the secure zone, and by consequence, a minimum performance of the isolated application. On the other hand, it also takes into account the current load of the platform when trying to dedicate more resources to the secure zone. The resources utilization rate is thus expected to be better than in a static approach thanks to dynamism. Moreover, untrusted applications are expected to be less penalized than in the static secure zone size scenario since less resources are expected to be dedicated.

— **Drawbacks:** However, this solution may penalize the isolated applications performance since achieving their maximum parallelism is not guaranteed. Finally, this solution requires fixing a minimum secure zone size for each isolated application. This can be determined considering the average application parallelism for example.

Resource reservation

Previous dynamic approaches may penalize isolated applications over non-isolated applications performance since isolated applications must wait for entirely idle contiguous clusters in order for their secure zones to be extended. An approach in order to cope with this when there are not enough available resources to create a secure zone of the size specified by the user (l in Algorithm 1), is to reserve currently non-available contiguous clusters in order to prevent allocating them to other applications once they are declared available again (i.e., released). This technique (Figure 3.6) favors the extension of secure zones over the performance of non-isolated applications. Indeed, when an application intended to be

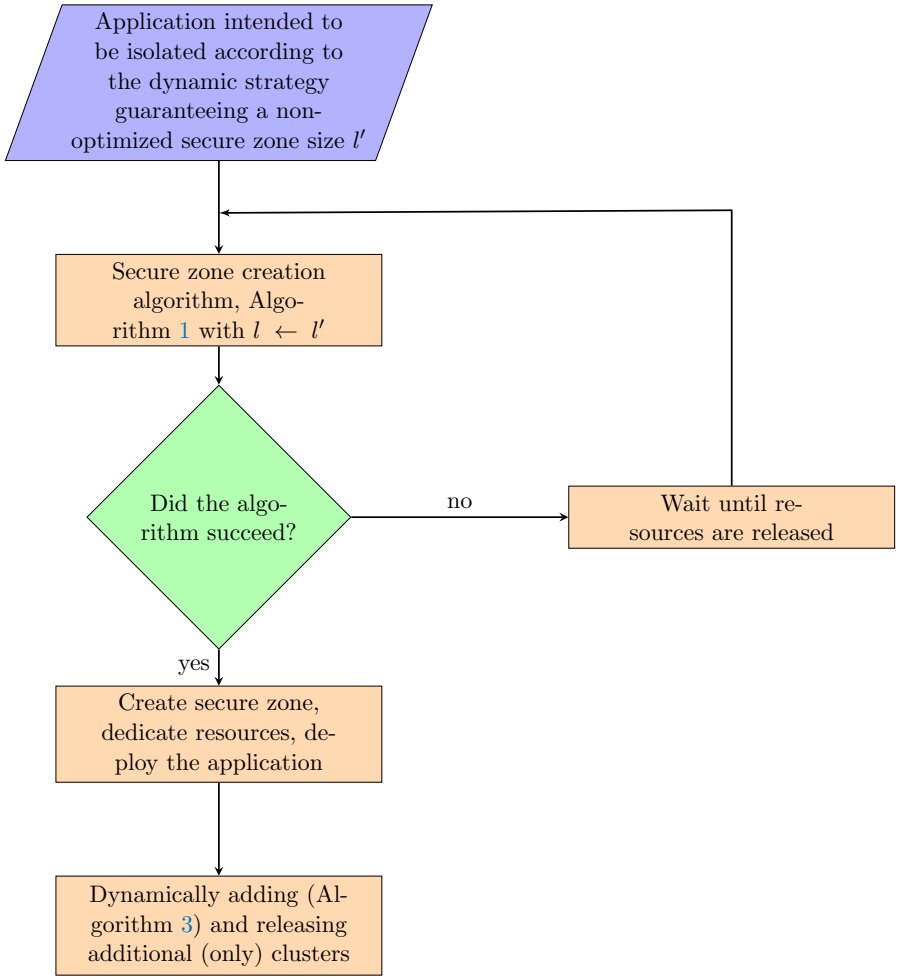


Figure 3.5 – Overview of the dynamic approach with a guaranteed non-optimized secure zone size approach

isolated is ready to be mapped and there are not enough available resources to create an optimal size secure zone, the largest available zone (which size is denoted $l' < l$) is chosen and dedicated to the isolated application which can start its execution. Further, the number of missing clusters from the secure zone ($l - l'$) are selected among contiguous clusters (not currently available) to be *reserved*. These latter need to be contiguous to clusters within the secure zone or if necessary to the reserved clusters. The selection of reserved clusters is done following the same principle as Algorithm 3. Reserved clusters are tagged and will no longer be allocated to other (trusted or untrusted) applications. As they are

not currently available for the isolated application, the secure kernel will constantly update the number of required clusters by the isolated application. Moreover, the secure kernel will monitor the state of the reserved clusters as well and when available, it will add them to the secure zone if still needed. Note that, excepting from original secure zone clusters, additional clusters (from the reserved ones) can be dynamically released from the secure zone if they are not longer needed. In the meanwhile, if there are not sufficient resources (computing or memory) within the secure zone, isolated application tasks will need to wait for resources within its secure zone to be released. In this case, if they finish by using the resources within its secure zone, less reserved clusters would be necessary. Consequently, the secure kernel may dynamically release clusters belonging to the actual secure zone as well as those tagged reserved clusters (not currently belonging to the secure zone).

As in this work application migration is not considered, once an isolated application starts to be executed, it cannot be migrated when a larger zone than its current secure zone is released. This choice is further discussed in Section 3.4.1.

Note that more sophisticated parameters can be used in order to decide which clusters are worthy to reserve. Indeed, the execution time left for processors in each cluster or the number of pending tasks could also be taken into account when selecting clusters to reserve. While this would entail higher and more complex activity on the secure kernel, it would certainly increase the chances of extending the secure zones.

— **Advantages:** In case of a *good* bet, this solution can be very interesting as isolated applications only need one single cluster to start to be executed and reserved clusters allow achieving a good performance of isolated applications. Furthermore, the dynamism of the approach entails *good* resources utilization rates.

— **Drawbacks:** On the other hand, if the bet turns out to be *bad*, this approach can be very penalizing for isolated applications. Indeed, if the reserved resources are not released during the execution of the isolated application, then, the secure zone will not be extended and new tasks within this application will wait for other tasks within the secure zone to finish in order to start its execution. Consequently, the isolated application may not achieve its maximum performance depending on the load of the platform and on the quality of the bet. Moreover, this approach requires a high activity on the mechanisms handling spatial isolation, compared to a static size approach.

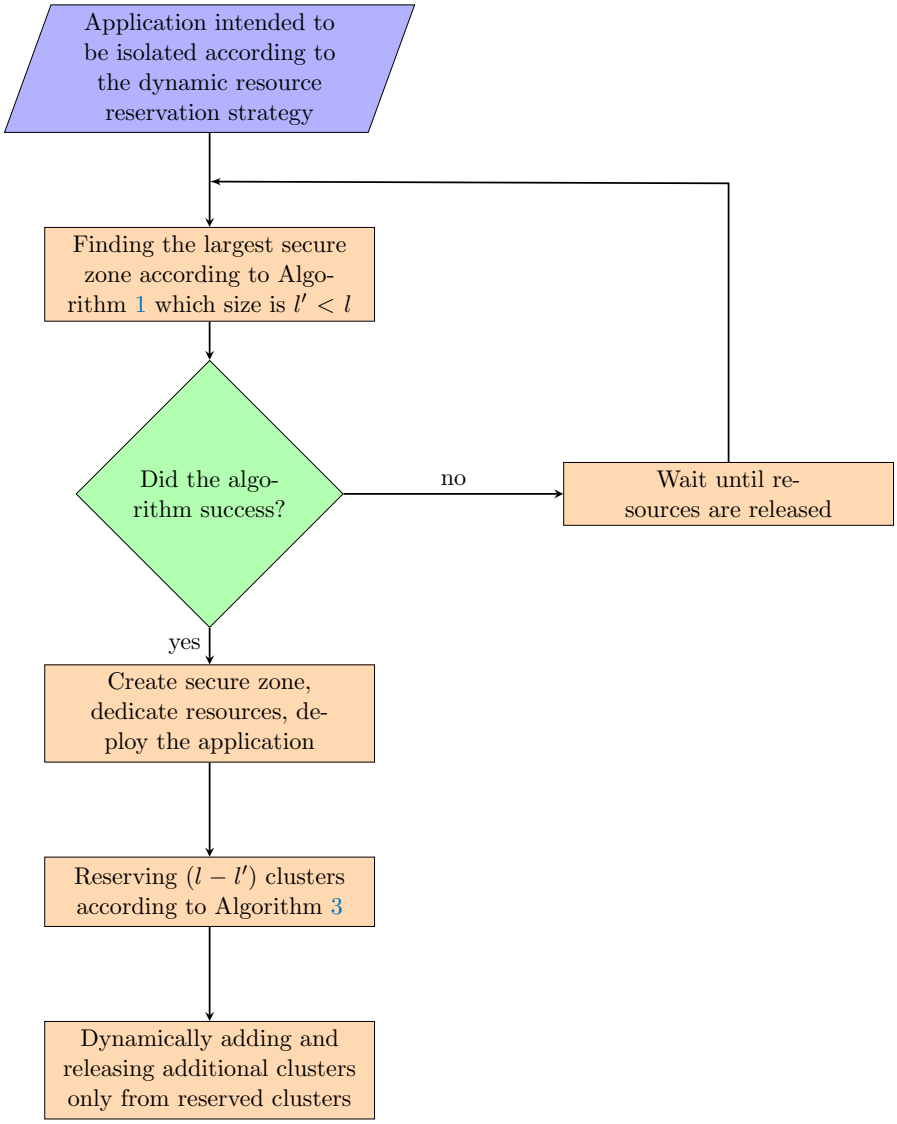


Figure 3.6 – Overview of the resource reservation approach

3.3 Summary of the proposed deployment strategies

This section summarizes some characteristics of the different deployment strategies proposed in this chapter. This summary is presented in Table 3.1. Each characteristic expected result is presented in a qualitative way. Note that these can change according to the load of the platform and execution scenario (e.g., number of isolated applications,

time at which applications are ready to be deployed, among others). Each expected result is accompanied with a corresponding color. For each characteristic, the lighter the color, the better.

3.4 Extension of the kernel services for spatial isolation implementation

To implement the proposed spatial isolation solution, we propose to extend the OS kernel services in order to integrate the spatial isolation enable mechanisms presented in this work. In this section, a recall of the threat model and the assumptions made for the implementation of this solution are first given. Then, the OS kernel services that have been modified are first presented in its original state (*baseline services*), then services extensions are explained.

3.4.1 Threat model and assumptions

As explained in Section 1.2.2, the physical platform is considered trusted. Moreover, it is considered that the platform is not physically accessible. Since this work focuses on cache-based SCAs, only attacks that do not require any physical access to the system to be performed are considered (i.e. logical cache-based SCAs).

In this work, the architecture is controlled by the ALMOS OS (see Section 1.2.1). This latter is responsible for the dynamic deployment of applications and management of resources. We propose to extend the OS kernel services in order to integrate the spatial isolation enable mechanisms. In this case, the kernel will be responsible as well for the secure deployment and execution of the spatially isolated applications. Consequently, the kernel services are required to be trusted and are part of the TCB. Therefore, we do not consider a full ALMOS OS, but a small ALMOS OS kernel including only the necessary services for the dynamic applications deployment and resources management. This assumption will make easier the evaluation of the proposed mechanisms as well. Finally, the kernel boot step on the system is considered protected [13].

The communication off-chip to memory or other peripherals is not taken into account, and it is assumed protected. Moreover, we assume that applications are independent from each other and thus communication between applications is not considered.

Furthermore, the NoC communications are assumed not to leak any information. In fact, in this work we address NUMA many-core architectures with logically shared but physically distributed memory (see Section 1.2.1). NoC attacks have not been proven practical

	Static	**Fully dynamic**	**Dynamic with a guaranteed size**	**Resource reservation**
Guarantee of isolated application performance?	guarantee	no guarantee	guarantee	no guarantee
Need to know about the application	yes	no	yes	yes
Overhead on the non-isolated applications performance	high	low	medium	medium
Large isolated applications waiting time before being deployed	high	low	medium	medium
Overhead on the resource utilization	high	low	medium	medium
High activity on the kernel services	medium	high	medium	medium

Table 3.1 – Summary of the different proposed strategies

in these architectures, therefore we do not consider the protection of NoC communications. However, even if the potential leakage of information through NoC communication on the targeted architectures has not been proven to be exploitable, countermeasures at the NoC level presented in Section 2.1.2, such as solutions proposed in [73], are compatible with our work and can be used as a complementary secure-enable mechanism. This point is further discussed in Chapter 6.

Moreover, for the implementation of this work, it is supposed that when launching an isolated application, it is specified if this one requires to be isolated and, according to the deployment strategy, the secure zone size if required.

Finally, application migration is not considered due to the induced complexity and cost. Indeed, migration here would include the secure remapping of the application and processor context switch as well as the memory remapping of the application data and instructions in order to leverage data locality. However, migration might be considered in the future in order to cope with problematics such as dark silicon, component aging, faulty components, etc.

3.4.2 ALMOS kernel services and integration of spatial isolation

In this thesis work, an ALMOS-like OS is considered [11]. The kernel services have been extended in order to integrate the spatial isolation enable mechanisms. In this subsection, each of these extended kernel services is first presented at its original state (i.e., baseline services in this work). Then, its extension in order to implement the proposed mechanisms is explained.

Monitoring service

The state of the platform needs to be constantly monitored in order to dynamically make decisions on the resource allocation. A tree-shaped structure (Distributed Quaternary Decision Tree (DQDT) [11]) is implemented to show the current state of each cluster at runtime.

In the TSAR [8] architecture, each cluster can encompass up to 4 processors. In this work we consider a 4-processor cluster architecture. In this monitoring structure, physical clusters (level 0 clusters in Figure 3.7) are grouped by 4 forming a logical cluster (level 1). In the same way, logical clusters are grouped by 4 forming an upper level logical cluster (level 2). Each physical or logical cluster is associated with a data structure (cluster structure

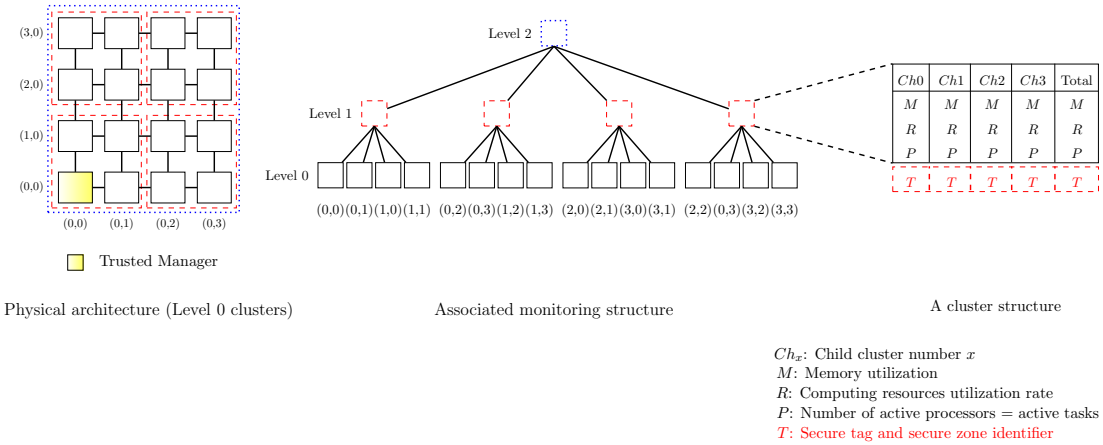


Figure 3.7 – Overview of the kernel monitoring structure

in Figure 3.7) containing some parameters describing the current state of the resources (processors and memory utilization rates as well as the number of active processors and tasks). For a physical cluster, the first 4 columns in its corresponding cluster structure concern the state of the memory and the computing resources within the physical cluster. The fifth column is the sum of the first 4 columns. On the other hand, for a logical cluster, the first 4 columns concern the states of the 4 child clusters (lower level clusters). The fifth column concerns the sum of the first 4 columns.

Figure 3.7 illustrates the monitoring structure for a 16 clusters architecture (level 0 in the figure). This organization has been designed for performance purposes. Indeed, the tree shape of this structure allows to make a decision both, globally and locally. First, a decision can be made globally by visiting the clusters monitoring structure from the root cluster (level 2 cluster in Figure 3.7) to lower level clusters (i.e., from top to bottom) until finding a solution. This prevents from visiting clusters that do not provide resources enough for the current request. Second, a decision can be made locally as well, in a bottom-up approach, in order to locally find available resources visiting the clusters structure from a given physical cluster (level 0). Consequently, the tree-shaped structure allows to make allocation decisions without visiting the totality of the clusters. In many-core architectures encompassing a wide number of clusters and processing resources, this improves the performance of allocation decision algorithms at the price of no guarantee of finding the global optimal solution each time. This monitoring shape is specially suitable for greedy algorithms.

The monitoring structure size (number of clusters) and shape is fixed and static according to the architecture characteristics. However, the current state parameters associated to each cluster (cluster structure in Figure 3.7) are regularly updated.

Two monitoring updates are implemented. A systematic update is required when a task is mapped on a processor or when a task has been executed and thus the processor is released, and when an application is finished and remaining processors and memory are released. For this update, it is necessary to visit and update values of the concerned physical clusters but also to visit each upper level parent (logical) clusters in a bottom-up fashion. A second update concerns resources utilization rates, these rates are periodically updated. The periodicity is configurable.

The kernel consults the monitoring structure every time a decision on resource allocation needs to be made. kernel services ensuring these decisions are presented below.

Extension: The monitoring structure has been extended in order to associate to each cluster an additional parameter (T in Figure 3.7) indicating when the cluster is dedicated to a secure zone as well as the secure zone identifier. For a physical cluster dedicated to a secure zone, the T value of each of the four first columns of its cluster structure is equal to the corresponding secure zone identifier. The value of T in the fifth column is equal to 1, indicating that this entire physical cluster is dedicated to a secure zone. For a logical cluster, the T value of each of the four first columns indicates the number of physical clusters in the corresponding structure branch that are dedicated to a secure zone. The T value in the fifth column is the sum of T values in the first four columns. These new parameters are taken into account when taking a resource allocation decision, since these clusters are not temporarily available to other applications until secure applications are released and tagged non secure again. Finally, when all the children clusters of one branch are tagged secure, the upper level cluster is tagged secure as well. In the same fashion, a parent status is updated when all its secure children clusters are released.

New application mapping service

The algorithm responsible to map new applications (i.e., the first task of an application), aims at finding an idle processor within a cluster with available memory in order to map the new task. For this, the mapping algorithm consults the monitoring structure, presented above, starting with the root cluster (level 2 cluster in Figure 3.7) in a top to bottom fashion until it finds a processor or until all the structure has been visited. The *available memory* is verified through a configurable memory available pages threshold.

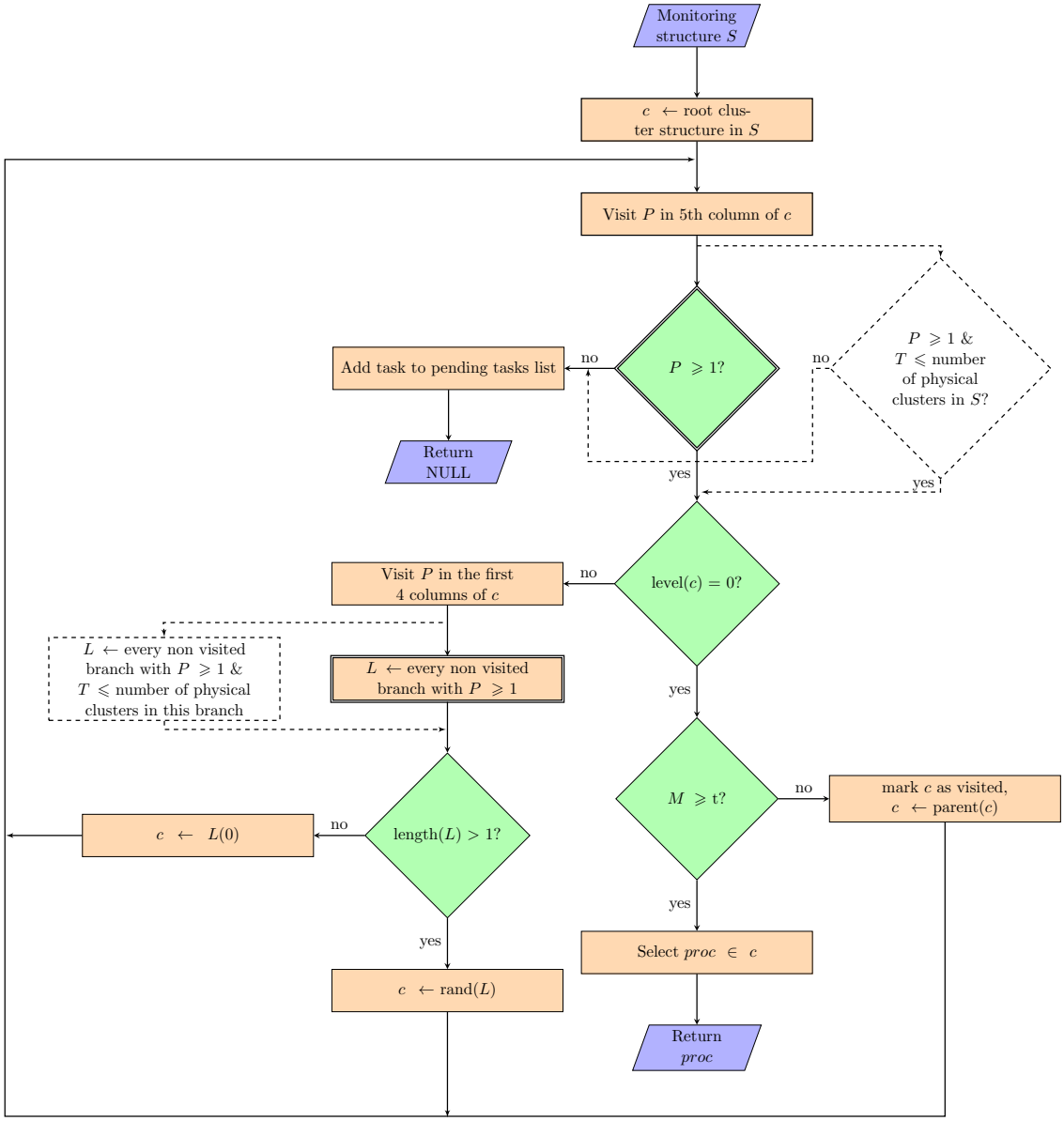


Figure 3.8 – Overview of original kernel application mapping algorithm and its extension in dashed lines (double line nodes are replaced by dashed line ones)

The mapping algorithm, aiming at finding an idle processor within a cluster with available memory starts by visiting the fifth column value of the root cluster architecture indicating the available computation P, and memory M resources on the entire platform (see

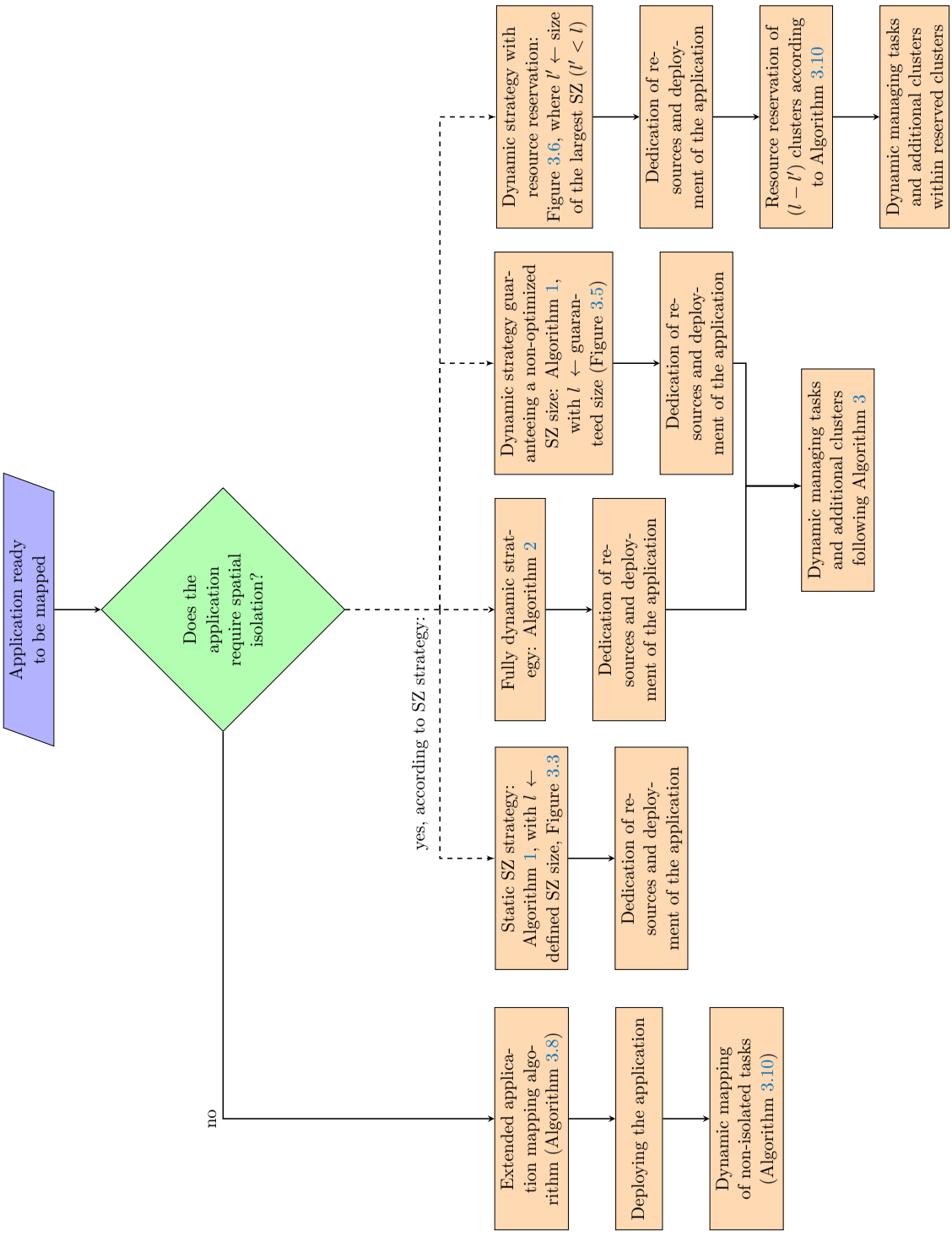


Figure 3.9 – Overview of the kernel functioning when mapping a new application

Figure 3.7). It verifies that there is at least one idle processor in the platform. If this is not the case, then the application is added to the pending tasks list and it will wait for a processor to be released. On the contrary, if there is at least one idle processor, then, the mapping algorithm will consult the first 4 columns which indicate which branch of the structure to visit. If several logical (or physical) clusters are candidates, one is randomly chosen in order to balance the load on the platform. This process will be repeated until an idle processor on a physical cluster (level 0 cluster) containing available memory is found. Then, the application is mapped.

Figure 3.8 shows the principle of the original kernel algorithm (and its extension presented below in dashed lines) for mapping new applications with the following notations:

— S: Monitoring structure (Figure 3.7),

— c: current cluster structure in S,

— P: number of idle processors in c,

— $proc$: selected idle processor,

— L: list of cluster candidates,

— M: available memory pages in c,

— t: memory pages threshold,

Extension: The main differences between the original and the extended application mapping algorithm are the following. First, it takes into account the extended version of the monitoring structure explained above which indicates if a resource is currently dedicated to a secure zone. In this case, the resource is temporarily not available to any other application (isolated or not). Second, applications, meant to be spatially isolated or not, are treated differently. A global view of the functioning of the kernel when mapping applications according to their requirements in terms of spatial isolation is illustrated in Figure 3.9. The figure makes reference of algorithms and figures explaining the different secure zone deployment strategies in Section 3.2, as well as the extended OS original services explained in this section. Note that, as explained in Section 3.4.1, we consider that, when deploying a new application, it is specified if the application is intended to be isolated or not. The kernel, according to this information is responsible for applying the corresponding mapping algorithm. On the one hand, if the application does not need to be physically isolated, then the original mapping algorithm (Figure 3.8) is used with the difference that resources within clusters tagged secure in the monitoring structure are considered temporarily not

available and cannot be allocated. This is illustrated in Figure 3.8, where double line nodes are replaced by dashed line ones. On the other hand, if the application to be mapped does need to be physically isolated, then, the algorithm for the creation of a secure zone corresponding to the considered deployment strategy is used. Different strategies for the creation and management of secure zone have been implemented. These strategies are presented in Section 3.2. Again, tagged resources (i.e., already dedicated to a secure zone) are considered temporarily not available. If a secure zone is created, its clusters are tagged secured with the application *ID* in the monitoring structure. Otherwise, if the secure zone could not be created, the application is added to the pending tasks list and will wait for available resources.

Finally, it can be noticed that in both cases, for isolated and non-isolated application mapping algorithms, the exploration space in the monitoring structure might be reduced since clusters tagged secure are not visited.

New task mapping service

A task may ask for the mapping of *child* tasks. A task and its child tasks are expected to communicate together by exchanging some intermediate data. Consequently, it is possible to select the node minimizing the distance between both tasks in order to minimize the communication costs which in turn favors performance. For this purpose, a child task must be mapped the closest possible to its parent task.

Similar to the new application mapping algorithm explained above, in order to take a mapping decision, this algorithm consults the monitoring structure. The main difference between these two mapping algorithms is that the new application mapping one searches for resources in the monitoring structure in a top to bottom approach, while the new task mapping algorithm visits the structure in a bottom-up fashion in order to take a decision locally from the cluster executing the parent task. Consequently, the physical distance between parent and child is minimized which minimizes the communication cost between them.

The algorithm searches for an idle processor starting from the physical cluster of the parent task (level 0 cluster). If no idle processor is found in this cluster, the algorithm goes up on the monitoring structure and searches on the logical cluster containing the parent physical cluster (level 1 cluster). The algorithm consults then the fifth column of the logical cluster structure and if there is at least one idle processor in one of the child clusters (level 0), it goes down to the found cluster. If there are several eligible clusters,

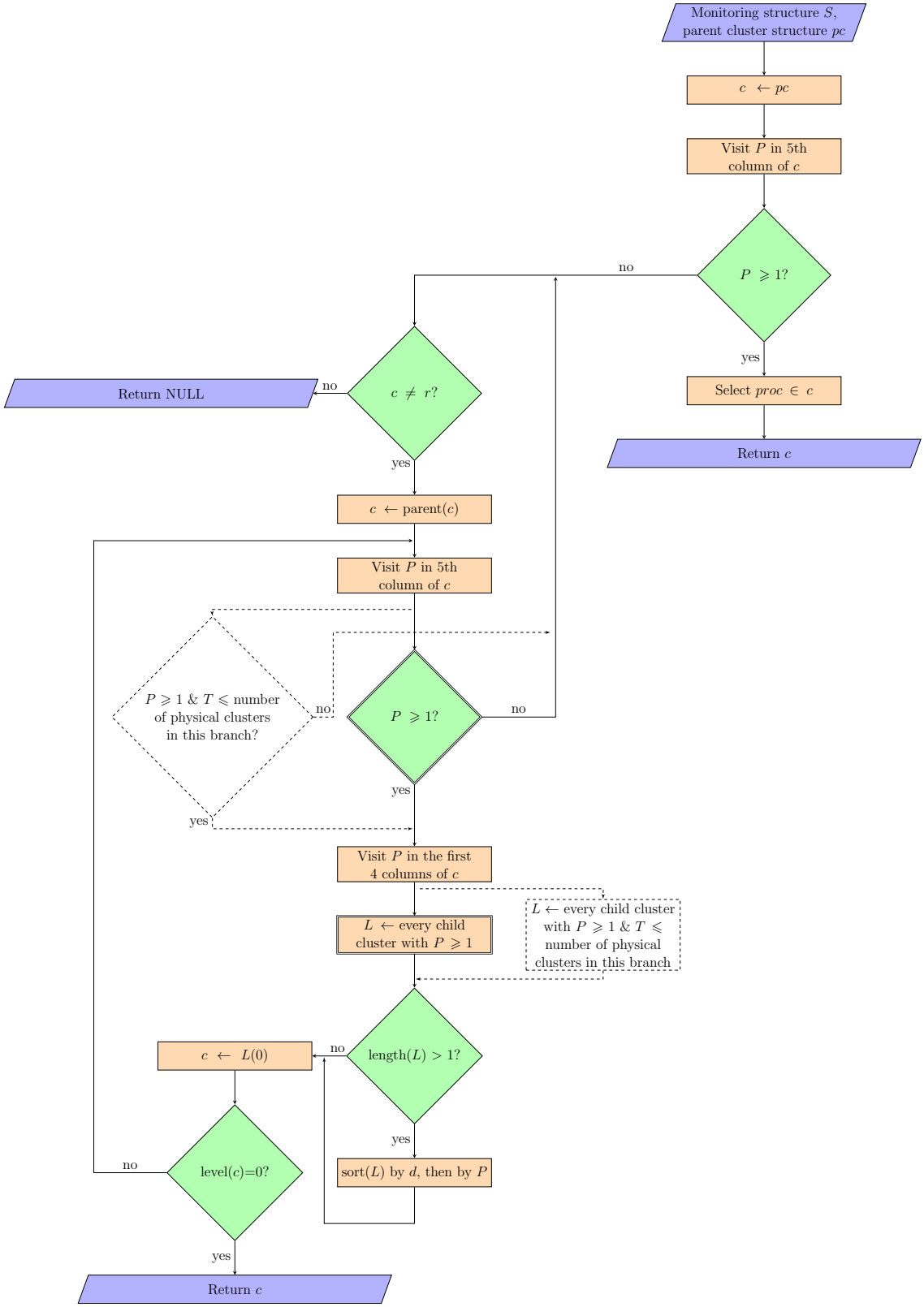


Figure 3.10 – Overview of original kernel task mapping algorithm and its extension in dashed lines (double line nodes are replaced by dashed line ones)

then a list of these clusters is sorted according to the distance between the eligible cluster and the parent task cluster first (in an ascending fashion), and according to the number of idle clusters on the eligible cluster in a descending fashion. Then, the first cluster on the list is selected. On the other hand, if there is no idle cluster, the algorithm goes up on the monitoring structure to the parent cluster (level 2 logical cluster). This process is repeated until either one idle processor is found, or until the entire structure is visited and no idle processor is found. In this last case, the task is added to the pending tasks list and it will wait for a processor to be released.

This algorithm is explained in Figure 3.10. Besides the aforementioned ones (see Section 3.4.2), it follows the following notations:

— *pc*: Parent cluster structure,

— *r*: Monitoring structure root cluster,

— *d*: distance between current *c* and the cluster of the parent task requesting for a child task mapping,

Extension: This service has been extended in order to take into account whether the new task belongs to a secure zone as well as the dedication of resources when consulting the monitoring structure. First, if the task requesting to map a child task does not belong to a physically isolated application, the kernel uses the original mapping algorithm (Figure 3.10) with the difference that the resources tagged secure are considered not available. This is illustrated in Figure 3.10, where double line nodes are replaced by dashed line ones. On the contrary, if the parent task application is physically isolated, the child task must be mapped within its secure zone. The algorithm starts by searching if the new task can be mapped within the same parent task cluster. If not, it tries to map it within the clusters belonging to its secure zone. If there is no idle processor within its secure zone, then there are two possibilities according to the considered secure zone deployment strategy. First, if the considered strategy allows to dynamically add resources to the secure zone (see Section 3.2), the kernel will try to extend the secure zone. Algorithm 3 explains the dynamic extension of a secure zone. If this is possible, the new cluster is tagged secure with the corresponding application identifier and the new task is mapped. If not, then the task will be added to the pending tasks list. The kernel will try to map it again when a resource is declared available. If on the contrary, the deployment strategy considers a static size secure zone and there are no idle processors within the secure zone, then the task to map is added to the pending tasks list.

Memory allocation

Memory availability is taken into account when taking a new application mapping decision. Once the application is mapped, tasks might request memory to the kernel. The memory allocation principle is similar to the task mapping algorithm presented above. In fact, in order to improve performance, ALMOS OS is designed to favor the communication between parallel tasks within an application. To do so, when a task asks for memory, the kernel will try to satisfy the request by allocating memory within the same cluster the request task executes. If this is not possible, the kernel will search for memory on a near cluster. The principle is similar to the algorithm illustrated in Figure 3.10, with the exception that: In the monitoring structure S, M is consulted instead of P and the parent cluster corresponds to the cluster on which the requesting task executes.

Extension: Memory is taken into account within the new mapping application algorithm presented above. A spatially isolated application does not share cluster resources with any another application. Therefore, memory allocated to an isolated application is within its secure zone. Additionally, according to the deployment strategy (see Section 3.2), the secure zone can be dynamically extended according to memory or processing requests. In this case, added clusters to the secure zone are required to be entirely available. On the other hand, memory pages within a secure zone cannot be allocated to any other isolated nor non-isolated application.

Clearing memory secure clusters service

Additionally, when a spatially isolated application is finished, memory within each of its secure zone clusters is cleared in order to prevent any leakage of information. Also, depending on the deployment strategy, secure zone clusters can be dynamically released, then memory within clusters is cleared before releasing the clusters and declaring them available again.

3.5 Summary of the extensions of the kernel services

This section presents a summary, through Table 3.2, of the different extensions of the kernel services explained above.

Kernel service	Extensions
Monitoring	— Extension of the monitoring structure with additional information indicating the dedication of clusters as well as the isolated application identifier — Different information for logical and for physical clusters
New application mapping	— Two different deployment strategies according to the status of the application (isolated or not) — Different deployment and management strategies for isolated applications have been implemented
New task mapping and memory allocation	— Different strategies according to the status of the task (belonging to an isolated application or not) as well as to the chosen deployment and management strategy if the task is isolated
Clearing of memory	— A new service has been added in order to clear remanent information after the execution of an isolated application

Table 3.2 – Summary of the extensions of the kernel services

3.6 Conclusion

In this chapter, the spatial isolation of sensitive applications has been proposed. This solution guarantees the protection of isolated applications against cache-based SCA. Several strategies for the dynamic deployment and management of secure zones have been explained. In order to implement these new mechanisms, the kernel services have been extended in order to integrate the proposed mechanisms responsible of secure zones and the new resources allocation constraints.

The prototyping environment and the evaluation of the proposed technique are presented in Chapters 4 and 5 respectively.

Chapter 4

Extension and exploration of MPSoCSim

Chapter contents

4.1 Motivation 74
4.2 MPSoCSim 77
4.2.1 Imperas/OVP technology 77
4.2.2 NoC simulator 77
4.3 Validation 80
4.3.1 Hardware implementation 80
4.3.2 Experimental protocol 81
4.3.3 Evaluation results 82
4.4 MPSoCSim extension 83
4.4.1 Clustering the architecture 83
4.4.2 Dynamic execution capabilities 86
4.5 Exploration with the extended MPSoCSim version 89
4.5.1 Exploration protocol 89
4.5.2 MPSoCSim available results 90
4.5.3 Exploration results 91
4.6 Conclusion 94

MPSoCSim is a simulator for NoC-based Multiprocessor Systems-on-Chip (MPSoCs). Based on the OVP [89] technology, it allows the evaluation of heterogeneous systems encompassing both, traffic generators as well as different OVP processor models including ARM processors and MicroBlazes among others. In this thesis work, this simulator has been extended in order to evaluate the contributions presented in Chapter 3. In this chapter, the motivation of MPSoCSim as well as its use in this thesis work are first presented in Section 4.1. Then, Section 4.2 introduces MPSoCSim in its original version and presents its validation through the comparison with a hardware implementation. Its extension in order to support complex clustered multi and many-core systems is explained in Section 4.4. Some results provided by the extended version of MPSoCSim are discussed in Section 4.5 in order to illustrate the capabilities of the simulator. Finally, Section 4.6, concludes this chapter.

4.1 Motivation

In order to evaluate the contributions presented in Chapter 3, a simulator enabling the validation of dynamic execution scenarios on multi and many-core clustered systems based on NoC is required. Additionally, modeled architectures are required to encompass private and shared resources. Moreover, running real applications composed of parallel tasks communicating through shared memory is required. Finally, simulations are required to be very fast in order to easily test different scenarios as well as application deployment and resource allocations strategies.

For this work, we chose to extend the MPSoCSim simulator [90] in order to meet our exploration requirements. In this section, an overview of existing simulators is presented and their comparison with MPSoCSim are given.

A well-known simulator for computer-system architecture research is gem5 [91]. This is a cycle accurate simulator specially suitable for processor microarchitecture exploration. However, gem5 does not currently support NoC communication infrastructure and provide cycle-accurate results at the price of very slow simulations (up to thousands of hours according to the simulated processor and application [92]). Therefore gem5 is not convenient for our study.

On the other hand, several existing simulators for NoC-based systems have been designed and focus on one subsystem of the multi or many-core architecture. Among these, some of them provide cycle accurate simulations focusing on the exploration of NoC design [93][94][95]. However, these simulators analyze the NoC with traffic patterns generated by traffic generators. This evaluation is essential since it allows the study of the NoC properties. Compared to these previous works, MPSoCSim is also convenient for NoC simulation and evaluation since it currently supports a parameterizable NoC, as well as traffic generators. However, in contrast with the work presented above, MPSoCSim also supports OVP processor models, enabling to run real application code.

SoCLib [15] is an open platform for virtual prototyping of multi-processors system on chip. This platform is composed of a library of SystemC simulation models for cycle accurate and bit accurate virtual components. TSAR architecture, considered in this work, has been modeled on this platform. However, the cycle accurate simulations come at the cost of very slow simulations. In contrast, this work proposes the exploration and comparison of different application deployment and resource allocation strategies through very fast simulations. Once the exploration is performed, the most interesting deployment strategies will be implemented in ALMOS on the TSAR architecture implemented on a SoCLib environment. This is further discussed in Chapter 6.

Similar to MPSoCSim, the work presented in [96] is also based on the OVP technology [89]. However, the authors in [96] focus on power estimations including an energy model in the OVPSim simulator. The simulation results are compared with a gate-level implementation of the simulated platform. On the contrary, power estimation is not the purpose of MPSoCSim nor of this thesis work. Table 4.1 summarizes the aforementioned simulators supporting NoC infrastructure.

Simulator	**Modeling language**	**Accuracy**	**NoC Topology**	**Processing elements**	**Simulation results**
Nirgam [93]	SystemC	Cycle accurate	Mesh, Torus, Butterfly	Traffic generators	Performance, power
Noxim [94]	SystemC	Cycle accurate	Mesh	Traffic generators	Performance, power
Booksim [95]	C++	Cycle accurate	Mesh, Torus, Butterfly	Traffic generators	Performance
SoCLib [15]	SystemC	Cycle accurate	Mesh	SoCLib processor models	Performance
Rosa et al. [96]	SystemC	Instruction accurate	Mesh	OVP processor models	Performance, power
MPSoCSim [90]	SystemC	Instruction accurate	Mesh	Traffic generators, OVP processor models	Performance

Table 4.1 – Overview of the existing simulators supporting NoC infrastructure

4.2 MPSoCSim

MPSoCSim is a SystemC simulator originally designed for the validation and exploration of NoCs [90]. OVP technology has enabled the attachment of OVP processor models for the modeling and evaluation of NoC-based MPSoCs. In this Section, MPSoCSim in its original version is presented.

4.2.1 Imperas/OVP technology

The OVP technology allows the connection of the OVP simulator to existing SystemC platforms [89]. Within the SystemC environment, the OVP simulator executes open source processors, memory and peripheral models in a completely instruction accurate way providing very fast simulations. An OVP model is provided with a TLM2.0 interface in the form of a C++ header file [97]. Amongst the execution of the processor model in a SystemC thread, SystemC instantiates a *tlmPlatform* object defining a quantum period. The quantum period sets how long each processor model instance waits before running again. Thus, it fixes the simulation scheduling quantum between the simulation of each processor. This quantum is adjustable. In MPSoCSim, its default value is 10μs.

4.2.2 NoC simulator

MPSoCSim is a SystemC simulator which first goal is to enable the NoC design exploration. It uses OVP capabilities in order to enable the attachment of OVP processors and peripheral models to the simulator and be able to model NoC-based distributed multiprocessor systems. In this subsection, the major components of the simulator are described.

Topology: Currently, this simulator solely supports 2D-Mesh topology. Figure 4.1 illustrates a 2-D Mesh modeled with this simulator.

Routing algorithm: Three different routing algorithms are implemented so far: XY, minimal west-first and adaptive west-first routing algorithms. However, new algorithms can be integrated.

Switching: The switching technique defines how the message units are moved from the router. In order to minimize the buffer depth of the routers, wormhole switching is used. In fact, in this flow control, each packet is broken into small pieces called flits. The first flit, called the header flit holds the network path for all the other flits in the packet (such as destination addresses). The result of using this switching technique is that messages are pipelined, and thus the buffer depth can contain one flit. Flow control is realized with

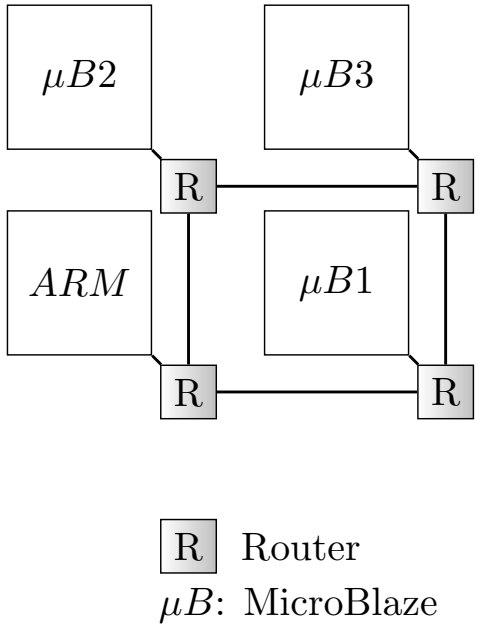


Figure 4.1 – 2D-Mesh NoC for MPSoCSim validation

acknowledge signals between the input buffers. In this simulator, the flit size amounts to 32 bits. The delay of a flit forwarded through a router (i.e., *Clock delay pass through*) is set to 1 cycle when no resource conflict occurs. In case of multiple messages trying to occupy the same output channel, only one message gets access to the requested output channel. Hence, the remaining messages are delayed.

Router: The router module is illustrated in Figure 4.2. It provides five connectors as MPSoCSim uses 2-D Mesh topology. Four to connect with the four possible directions routers and a local port connecting the router to the corresponding Network Interface (NI). Each connector consists of one initiator and one target sockets to enable TLM data transmission. Target sockets encompass one FIFO for saving incoming flits. As explained above, as wormhole switching is considered, the FIFO depth is set to 1. MPSoCSim uses a round robin arbitration in order to assign the output socket among the input sockets requesting the output.

Network Interface: While routers handle the communication through the NoC, a SystemC TLM NI connects each router to a local group called *node* encompassing the processing element. Figure 4.3 illustrates an MPSoCSim node. The NI consists of one

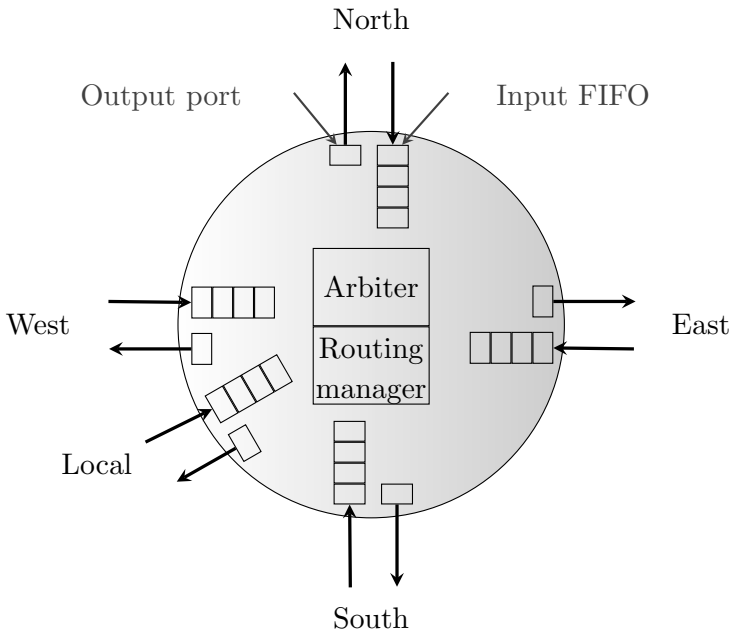


Figure 4.2 – MPSocSim router structure

initiator and one target socket for the connection with the router, and one initiator and one target socket for the connection with the processing element. NIs are addressed via a local bus as any other peripheral.

Processing elements: MPSoCSim supports traffic generators as well as Imperas/OVP processor models. Traffic generators periodically send messages to the network. Periodicity is configurable. Traffic generators enable the functionality and performance evaluation of the NoC design. Additionally, OVP processor models (ARM, MicroBlazes, MIPS32, etc.) can be attached to the nodes in order to be able to run real application code. Each node can be attached to a different OVP processor enabling the simulation of heterogeneous systems.

Memory: MPSoCSim first version supports distributed memory. Each processor can access by reading and writing to its own cluster *local RAM* (see Figure 4.3). In order to access to data stored into other local RAMs, data is sent through the network.

Additional timer module: Peripherals can be attached to local buses within clusters. In MPSoCSim, a timer module is attached within each cluster in order to enable the readout of execution time for applications.

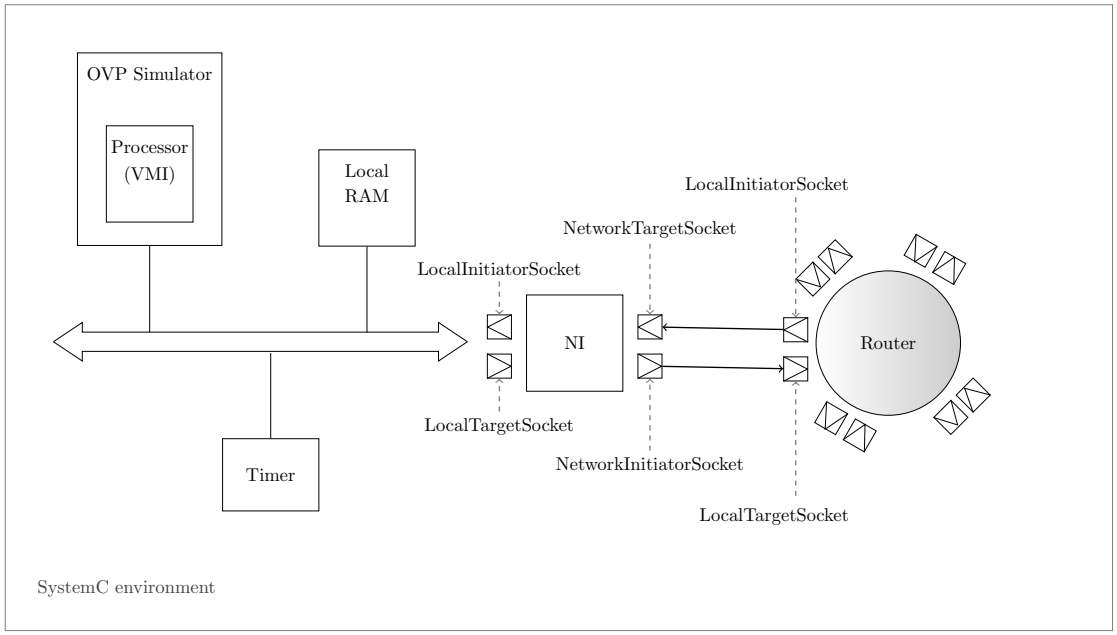


Figure 4.3 – MPSocSim simulator overview

MPSoCSim has been validated through the comparison of a simulated MPSoC with its hardware implementation. These experiments are shown in next section.

4.3 Validation

In [90], an MPSoC has been both, simulated with MPSoCSim, and implemented on a Xilinx Zynq device [98] in order to evaluate MPSoCSim accuracy. This section summarizes the hardware implementation and experimental protocol used for the MPSoCSim evaluation. Then, the evaluation results are illustrated.

4.3.1 Hardware implementation

MPSoCSim has been evaluated in its original version in [90]. For this evaluation, a 2×2 2D-Mesh NoC is considered. One node encompasses an ARM processor while the others encompass one MicroBlaze. This system, illustrated in Figure 4.1, has been implemented on a Xilinx Zynq device [98] which provides an ARM processor besides an FPGA. In the implemented system, the FPGA contains a NoC and 3 MicroBlazes. For this comparison,

Parameters	Chosen value
Quantum period	10μs
ARM Frequency	667MHz
MicroBlaze (μB) Frequency	100MHz
Nominal MIPS	100
Real flit time (ARM)	850ns
Real flit time (μB)	40ns
Clock delay pass through	1
Network frequency	100MHz
Network topology	2-D Mesh
Network routing algorithm	XY
Network size	2×2 clusters

Table 4.2 – System parameters used for evaluation in MPSoCSim

only one core of the ARM processor is used. The ARM processor communicates via the high performance port (HP) to the NoC. The processors are connected to each other by RAR-NoC [99] in a 2D-Mesh topology. The MicroBlazes are linked via the FSL interface to the NoC. Packets are sent following XY algorithm using wormhole routing. Finally, the frequency of the FPGA is set to 100 MHz and the ARM processor runs at 667 MHz.

4.3.2 Experimental protocol

Simulation in MPSoCSim: For the comparison presented above, the considered system is simulated using MPSoCSim. Table 4.2 shows the simulation parameters configuration chosen for this evaluation in order to fit the hardware implementation.

Applications considered for the MPSoCSim evaluation: For evaluation, benchmarking applications taking advantage of the NoC capability of the simulator have been developed. Tiled matrix multiplications have been considered for the comparison of the simulator and the hardware implementation results [90]. In this application, four tasks run in parallel. A *master task* running on the ARM creates the matrices to be multiplied and sends the necessary rows and columns to each of the three *slave tasks* running on the MicroBlazes by dividing the matrices into equal parts. Each MicroBlaze task does the corresponding computation and sends the results back to the ARM task which, after its computation, collects all the results. Figure 4.4 illustrates the split of data and computation load for the multiplication of matrices A and B where C is the results matrix.

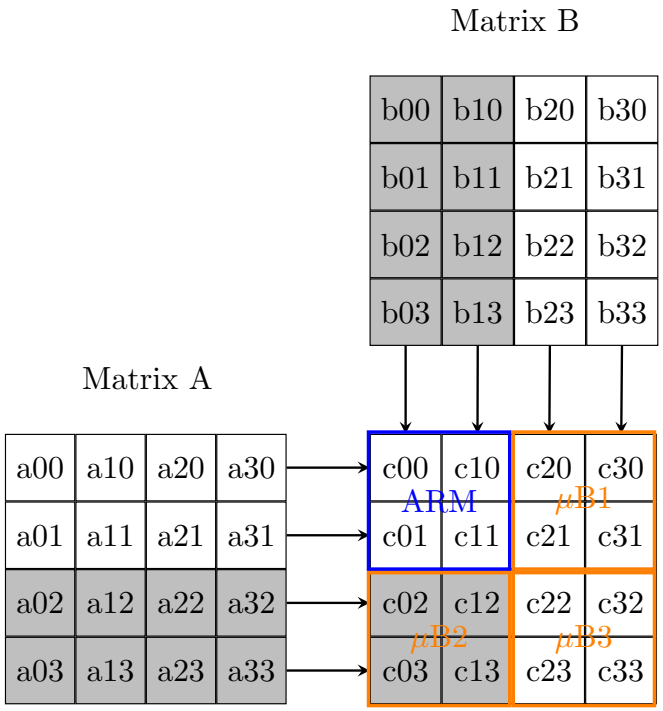


Figure 4.4 – Tiled matrix multiplication [90]

4.3.3 Evaluation results

In order to evaluate MPSoCSim, an MPSoC has been simulated with MPSoCSim and has been implemented on a Xilinx Zynq device [98]. The same above-presented application has been executed in both environments. In this subsection, execution time results (simulated execution time in MPSoCSim, Section 4.2) on both are presented and compared. Execution time results are illustrated in Figure 4.5 for standard tiled matrix multiplications for 4×4, 8×8 and 16×16 matrices.

Results are compared in terms of the *relative deviation*, which is the simulated execution time in relation to the execution time on the hardware implementation. Results show a maximum relative deviation of 17,4% for the 4×4 matrix multiplication. However, this deviation decreases for higher matrix multiplication sizes and results in a relative deviation of 2,5% for the multiplication of 16×16 matrices. These results show how accurately MPSoCSim models the actual hardware implementation.

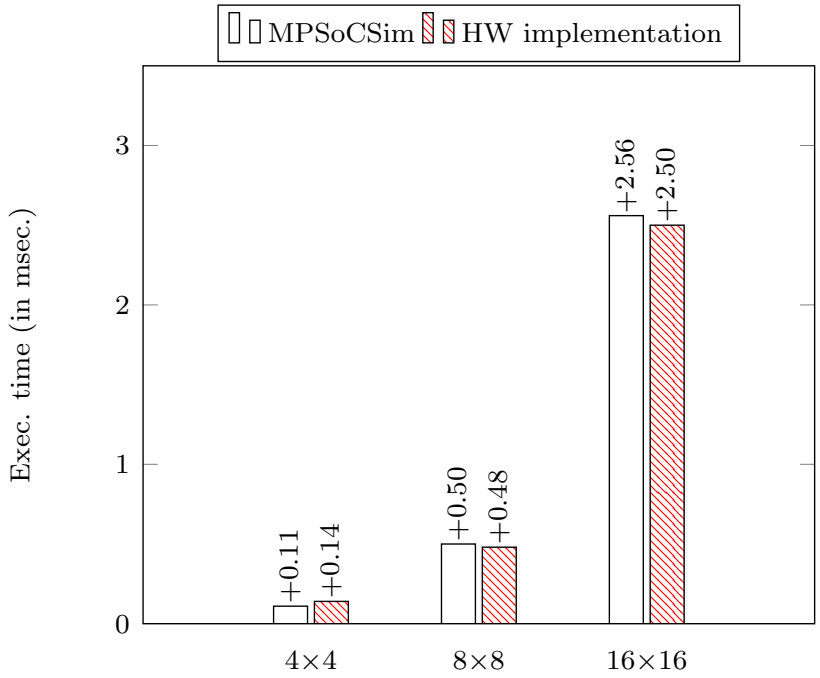


Figure 4.5 – Comparison between MPSoCSim and the hardware implementation [90]

4.4 MPSoCSim extension

In this section, the extension of MPSoCSim is first explained. Then, its capabilities for modeling a dynamic scenario are discussed.

4.4.1 Clustering the architecture

In its original version, MPSoCSim, (designed for NoC exploration) was able to simulate simple multiprocessor architectures, with distributed memory and where each node included one single processor. In this thesis work, MPSoCSim has been extended, in order to allow the modeling of more complex clustered multi and many-core systems supporting distributed memory between clusters and shared memory within a cluster.

The extended version of MPSoCSim allows each cluster to encompass several processors, private and shared resources. For this, and in order to keep the simulator flexible, an entity called *subgroup* has been implemented (dashed blue line in Figure 4.6). Each subgroup encompasses a processor and a private memory. The processor is directly connected as master to a *local bus*. This bus is a generic decoder of class memory mapped *TLM decoder*,

provided by OVP. Each processor can access by reading and writing to its own private memory. Additionally, a memory-mapped bridge between the local bus and a *shared bus* maps shared resources enabling the processor to access to resources shared among all the processors within the cluster (e.g. memory and peripheral). On the contrary, this solution prevents processors to see or access to resources private to other processors. The template of the local bus is *decoder <int NR_OF_INITIATORS, int_NR_OF_TARGETS>*. Initiators correspond to components able to initiate a transaction on the bus, while the targets correspond to the components answering to transactions coming from the bus.

In the case of a subgroup, the local bus instantiation is *decoder <2,2>*. The two bus target sockets are connected to initiator instruction and initiator data of the processor. One bus initiator socket is connected to the target socket of the private memory. Finally, the second initiator socket is connected to a target socket of the shared bus. Initiator and target sockets of the local bus are illustrated for the highlighted subgroup in Figure 4.6.

Resources shared between all the processors within the cluster are mapped to the shared bus. This bus is instantiated according to the number of subgroups and shared resources. In this work, we consider a TSAR-like architecture where each cluster is composed of four subgroups, a shared memory, and a network interface, as illustrated in Figure 4.6.

The shared bus instantiation in this case is *decoder <5,2>*. Four out of five initiators correspond to the four subgroups considered in this cluster, these are connected to four bus target sockets. The last target socket is connected to the initiator socket of the NI. The NI has a target socket as well, since processors car start as well transactions between them and the NI. This target socket is connected to one of the two bus initiator sockets. Finally, the last bus initiator socket is connected to the target socket of the shared memory.

The interest of including private and shared memory is twofold, performance and privacy. In fact, private memories are not accessible by other processors. This allows each processor to store and run independent code close to the processor. Moreover, intermediate results can be stored in the private data in order to favor the locality of accessed data for better performance. On the contrary, shared memory enables the communication between processors. In fact every processor within the cluster can access by reading and writing to the shared memory. Finally, processors use this memory in order to communicate with distant processors within distant clusters. In this work, the considered memory sizes are 3 MBytes and 64 KBytes for shared and private memory respectively.

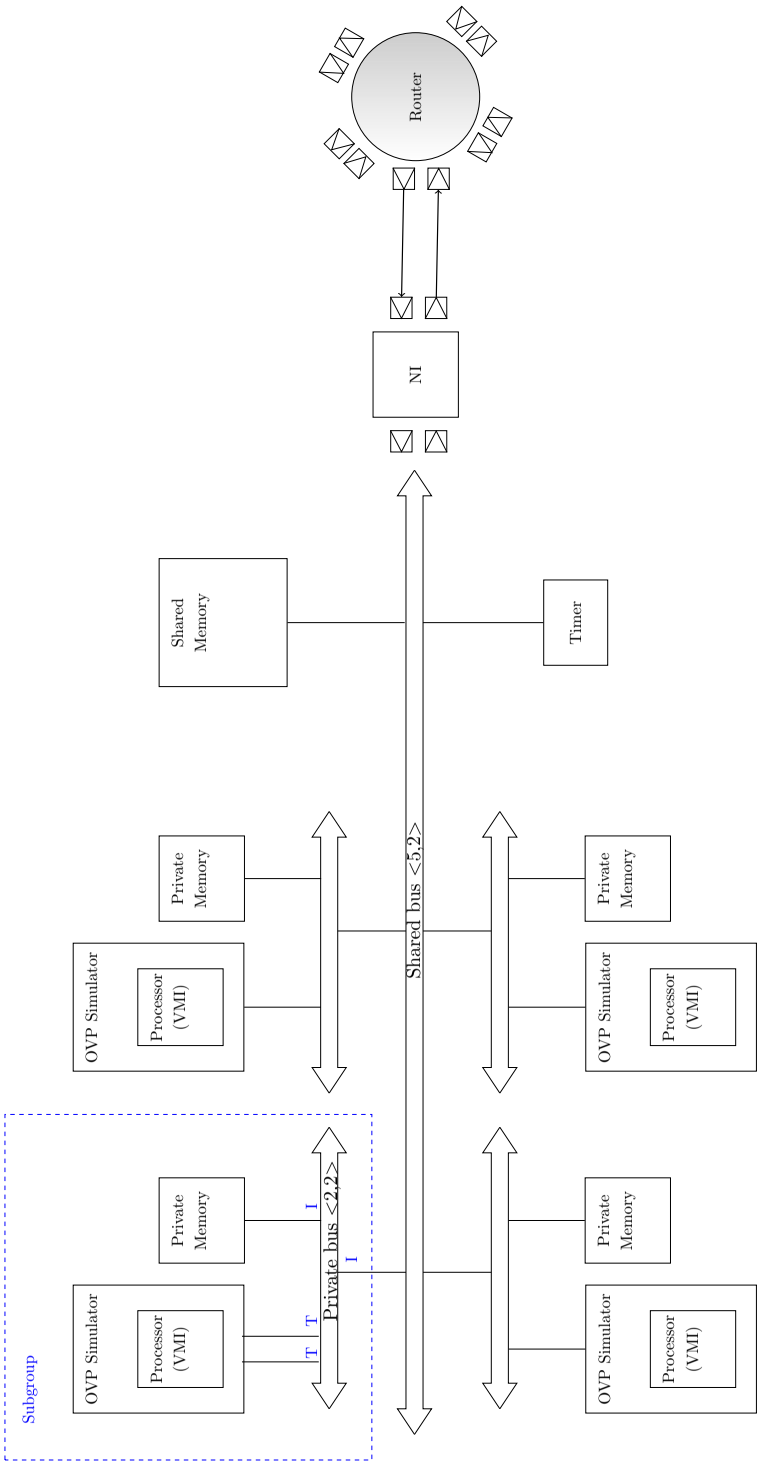


Figure 4.6 – Overview of the extended MPSoCSim

```
icmTLMPlatform                platform;
MCA_NoC_Simulator::Network    network;
ram*                          privateMem[(xSize*ySize)*4];
ram*                          sharedMem[xSize*ySize];
arm*                          armCPU;
decoder<2,2>*                 armLocalBus;
decoder<2,3>*                 armSharedBus;
microblaze*                   microblazeCPUs[((xSize*ySize)*4)-4];
decoder<2,2>*                 microblazeBus[((xSize*ySize)*4)-4];
decoder<5,2>*                 sharedBus[(xSize*ySize)];
MCA_NoC_Simulator::Timer*     timer;
```

Figure 4.7 – Considered platform instantiation

The solution proposed in this work for the extension of MPSoCSim allows the flexibility of the simulator modeling. In fact, each cluster is composed of a configurable number of *subgroups*. Moreover, each cluster design is independent. Consequently, each cluster can encompass a different number of subgroups (i.e. processors). Finally, within a cluster, subgroups are independent and can be different from each other (i.e. MIPS, ARM, MicroBlazes, among others).

Notice that other peripherals can be attached both, to the shared bus or to private buses (e.g. aforementioned timer module).

Figure 4.6 illustrates an extended MPSoCSim cluster composed of a shared memory and timer as well as four subgroups, each including a processor and private memory.

Figure 4.7 shows the instantiation of components in a platform of 4×4 (where xSize and ySize are equal to 4), where 14 regular clusters are composed of 4 subgroups instantiated with a MicroBlaze processor, and 1 cluster composed of one single subgroup with an ARM processor. Finally, this last cluster includes an additional timer module attached to the shared bus *armSharedBus*. Thus, in this case, the shared bus includes one additional initiator socket (*d*ecoder <2,3>).

Figures 4.8 and 4.9 show the parameters for the instantiation of the two considered processors.

4.4.2 Dynamic execution capabilities

The simulator benefits from the OVP processor models which support different OSes such as Linux and FreeRTOS. Furthermore, some other OVP and SystemC features can be used in order to simulate a dynamic scheduling of applications. Indeed, the SystemC

```
icmAttrListObject *attrsForARMCPU() {
    icmAttrListObject *userAttrs = new icmAttrListObject;
    userAttrs->addAttr("endian",        "little");
    userAttrs->addAttr("compatibility", "nopSVC");
    userAttrs->addAttr("variant",       "Cortex-A9UP");
    userAttrs->addAttr("UAL",           "1");
    userAttrs->addAttr("mips",          "100.0");
    return userAttrs;
}
```

Figure 4.8 – OVP ARM processor instantiation parameters

```
icmAttrListObject *attrsForMicroblazeCPU() {
    icmAttrListObject *userAttrs = new icmAttrListObject;
    userAttrs->addAttr("endian",        "little");
    userAttrs->addAttr("mips",          "100.0");
    return userAttrs;
}
```

Figure 4.9 – OVP MicroBlaze processor instantiation parameters

environment enables to run the simulator for a given period of time. This feature enables to run, stop and resume the execution. Moreover the OVP environment allows the simulator to reset any processor through its reset pin. In this way, the processor's program counter is reset and the application on this processor can be restarted from the beginning or another application code can be executed. It is worth noticing that during the time taken to reset the first processor, the other processors are still running independently. These 2 mechanisms can be used together in order to control the dynamic execution on the platform.

Depending on the requirements of the simulation, two cases are possible.

First, let's consider two processors (*P1* and *P2* in Figure 4.10) running in parallel executing the same *code 1* (previously loaded in each private memory). When the simulation starts (i.e. *sc_start(time)* in SystemC language), the code on both processors starts at the point *a*. The simulation runs for a specified time *time*, the code on both processors stops at point *b*. When the simulation stops, processor P2, for example, can be reset which in consequence will reset its programming counter to the beginning of its code (i.e., a). When the simulation starts again, the code on processor P1 restarts at the last point (point b) but processor P2 restarts at point a. In this case, the reset time of processor P2 (t_{reset}) is not taken into account on the simulation time. This first case is illustrated in Figure 4.10.

Second, the reset time and reload of code on processors (i.e., $t_{reset+reload}$ in Figure 4.11)

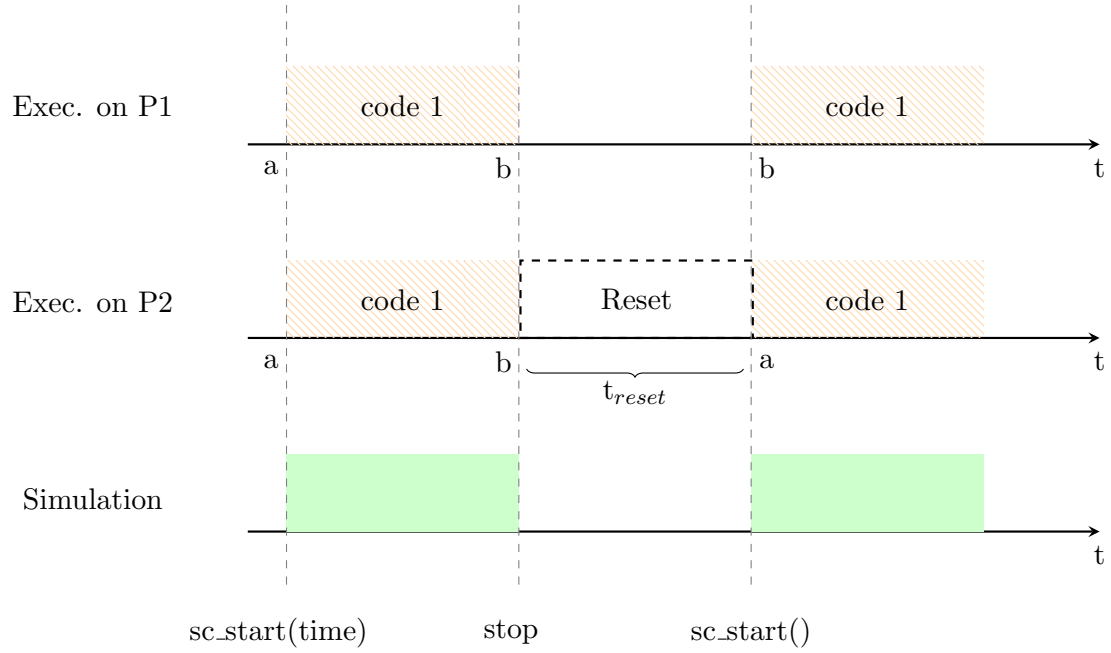


Figure 4.10 – First possible case of dynamic execution simulation

can be taken into account on the simulation execution time. Now consider the same scenario in which two processors P1 and P2 are running in parallel executing the same code (code 1). At some point during the simulation, P2 can be reset and a second code (e.g., *code 2*) can be loaded. Notice that in this case, the simulation has not been stopped during the reset and reload of code. Consequently, P1 is still running its own code. This is illustrated in Figure 4.11.

These capabilities are used in order to execute a dynamic execution scenario in which one processor is able to act as the manager on the platform and to control the deployment and execution on the system.

In the next section, the capabilities of the extended version of MPSoCSim are explored through different execution scenarios.

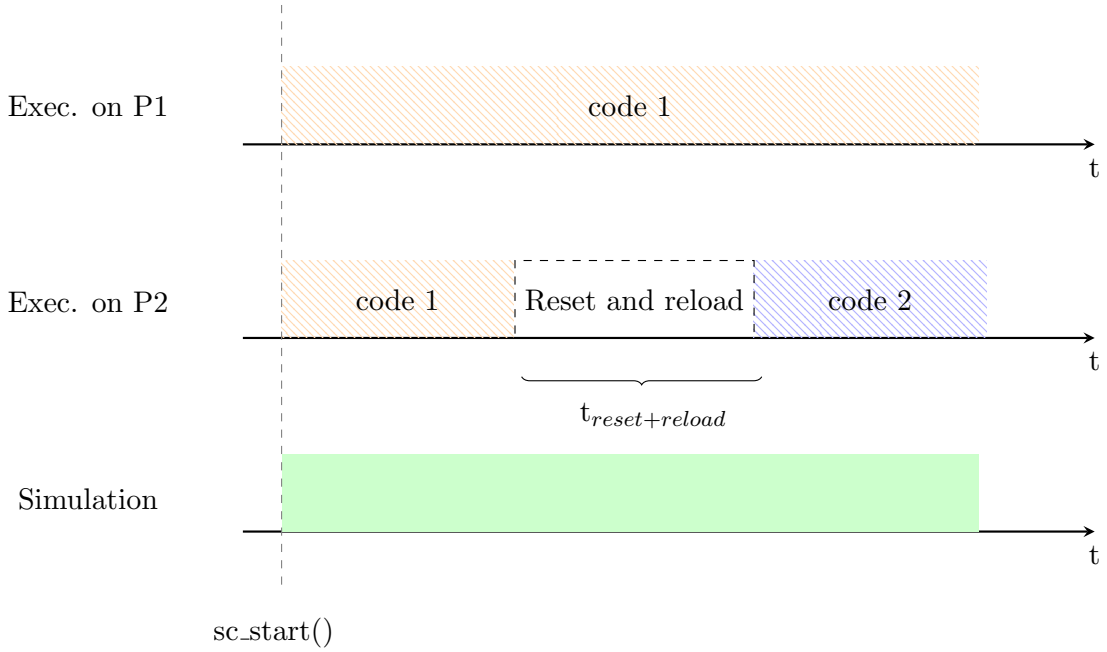


Figure 4.11 – Second possible case of dynamic execution simulation

4.5 Exploration with the extended MPSoCSim version

MPSoCSim has been extended in order to be able to model complex clustered multi and many-core architectures including shared and private resources. The extension of this tool provides different results for the evaluation and comparison of different architectures as well as of different execution scenarios on the same architecture. In this section, some experimentation results provided by the tool are presented. First, the experimental protocol is given. Then, results available with MPSoCSim are explained. Finally, exploration results are presented.

4.5.1 Exploration protocol

In this subsection, the experimental protocol for the MPSoCSim extension exploration is presented.

First, the presented extension of MPSoCSim relies on the validation of MPSoCSim in its original version [90] (Section 4.3.3). Therefore, for experiments on the extended MPSoCSim, the same system parameters considered for the validation of MPSoCSim are

used. These are summarized in Table 4.2.

Further, the same applications as in Section 4.3.2 have been used for this exploration. Considered matrices sizes vary from 2×2 to 64×64.

Finally, two different architecture sizes are considered. First, a 2×2 cluster architecture (3 clusters encompassing 4 MicroBlaze processors each and one cluster encompassing one ARM processor) is considered. Then, the same experimentations are performed on a 4×4 cluster architecture encompassing in total 60 MicroBlazes and one ARM processor.

4.5.2 MPSoCSim available results

Several results concerning both, execution and system simulation, are provided by MPSoCSim and have been explored after the simulator extension presented in the above section.

- **Network interface communication:** MPSoCSim provides some communication results in terms of number of received and sent packets to each network interface as well as the current and maximal data rates.
- **Simulated execution time:** It is the time given by the aforementioned timer module during execution. This value specifies the time that is needed for an application or a portion of code to be executed excluding the time of initialization of processes. The deviation between this value and the execution time on the real hardware shows the accuracy of the simulator.

Besides these results, other OVP specific values are also provided.

- **Number of instructions:** The total number of simulated instructions as well as per processor are available.
- **User time:** OVP provided execution time is the time spent for the execution of instructions on the host (user) machine.
- **System time:** This value indicates the time spent by the host machine to execute the instructions of the simulated process.
- **Elapsed time:** This value corresponds to the time from the beginning to the end of the simulation.

These results are presented for the considered execution scenarios (Section 4.5.1) in the next subsection.

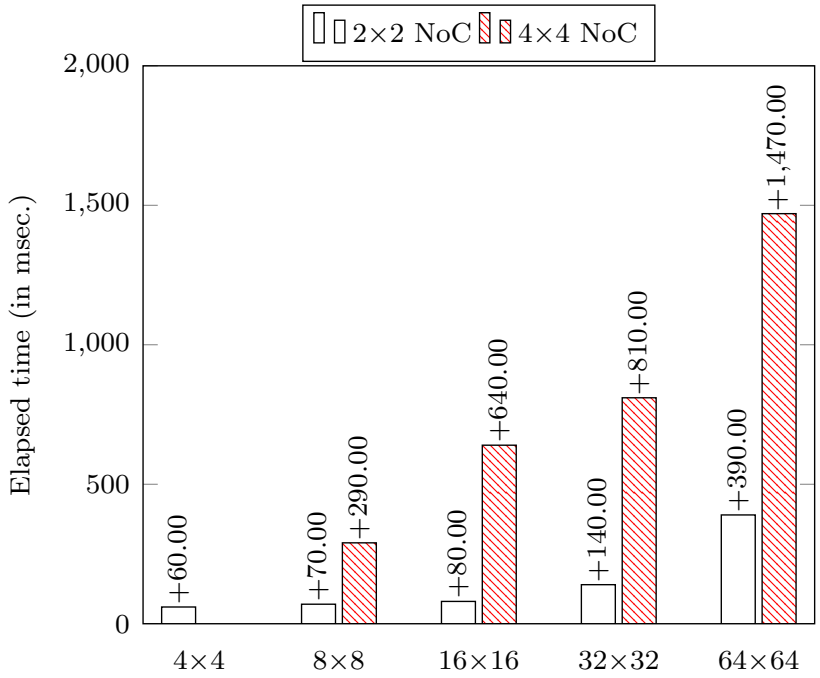


Figure 4.12 – **Elapsed time** (in msec.) for different sizes of matrix multiplications on 2×2 and 4×4 NoC clustered architectures (12 MicroBlazes and 1 ARM, and 60 MicroBlazes and 1 ARM respectively)

4.5.3 Exploration results

First, Figure 4.12 shows the evaluation on the **elapsed time**, in milliseconds, necessary on the host machine to simulate the execution of several sizes of matrix multiplications on the two considered platforms; a 2×2 cluster and a 4×4 cluster architectures (12 MicroBlazes and 1 ARM, and 60 MicroBlazes and 1 ARM, respectively). The elapsed time on the host machine increases with the size of the simulated platform, however it can been noticed that the simulator enables very fast simulation time on the host machine even for the most complex architecture tested in this work (around 1.5 seconds for a 64×64 matrix multiplication on a 60 MicroBlazes and 1 ARM architecture).

Then, **total simulated execution time** results, in milliseconds, are gathered in Figure 4.13. These results are useful, for instance, in order to choose the best architecture size for a given execution scenario by comparing the trade off of having more computational resources and the communication complexity overhead induced for larger and more

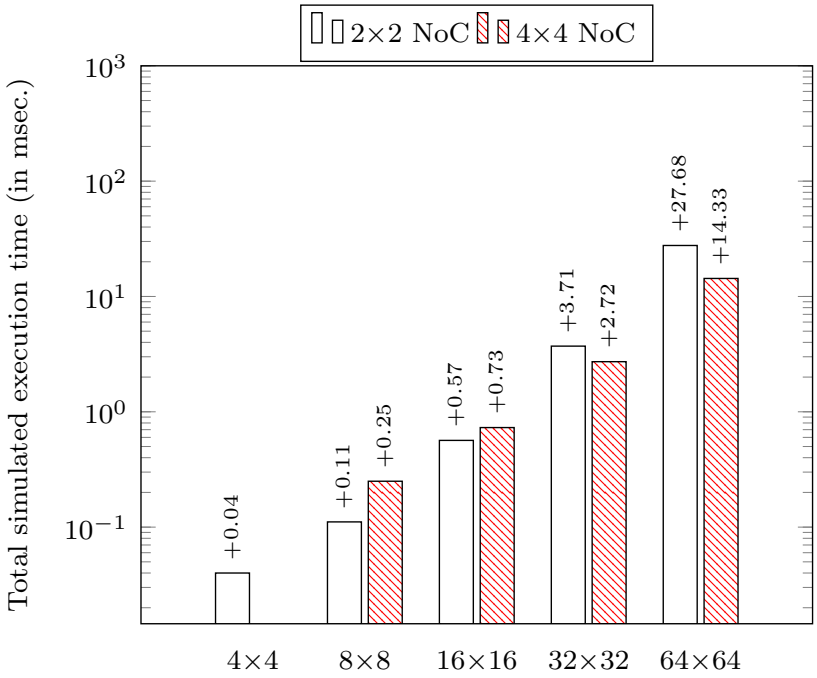


Figure 4.13 – **Total simulated execution time** (in msec.) for different sizes of matrix multiplications on a 2×2 and 4×4 NoC clustered architectures (12 MicroBlazes and 1 ARM, and 60 MicroBlazes and 1 ARM respectively)

complex platform. In this case, for example, a straightforward remark is that, according to the results in Figure 4.13, for small matrix multiplications (below 32×32 matrices multiplications), it is not worth having a bigger platform since the computation is negligible compared to the communication cost. In fact in bigger platforms, the packets, always sent by the ARM processor task, need to cross a greater number of routers to get to their destination, the MicroBlaze processor tasks polling the memory until the packets arrive. On the other hand, for greater computation, it is worth executing the application on bigger platforms. However, if we look deeper into results, it can be noticed that executing a 16×16 matrix multiplication on a 4×4 cluster architecture provides a gain of 22% of the execution time compared to the execution on a 2×2 clustered architecture. However, to obtain this gain, 60 MicroBlazes are required instead of only 12. Thus, the results obtained with this simulator can be used in order to choose the best size of the platform for a given execution scenario.

Finally, results in terms of **simulated instructions** for a given execution scenario,

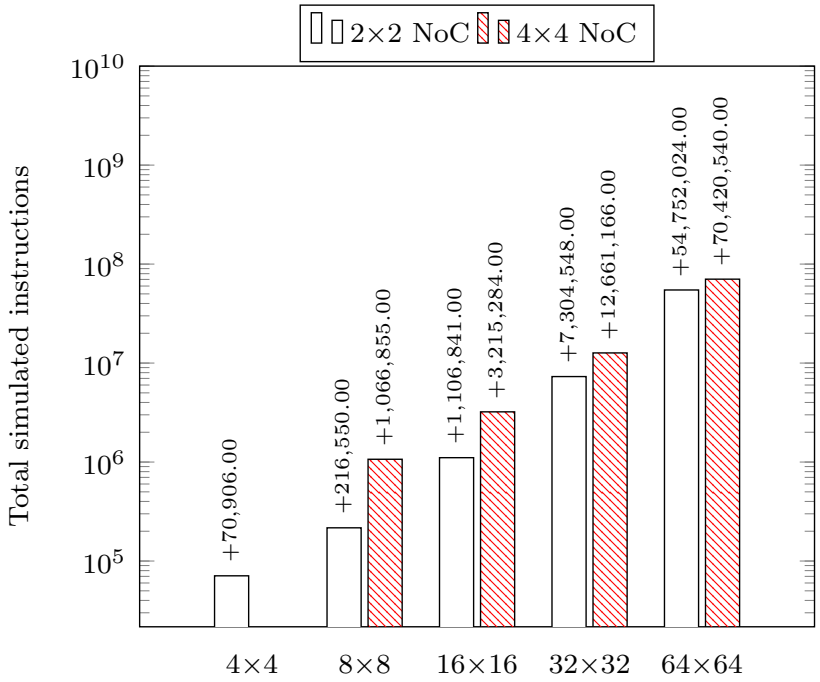


Figure 4.14 – **Total simulated instructions** for the execution of different sizes of matrix multiplications on 2×2 and 4×4 NoC clustered architectures (12 MicroBlazes and 1 ARM, and 60 MicroBlazes and 1 ARM respectively)

are presented in Figure 4.14. These results allow the evaluation of the communication cost for different platforms on a given execution scenario. In fact, it can be seen that the communication overhead for a larger platform decreases when the matrix multiplication size increases for the same computation load (down to 22% more instructions for a 64×64 matrix multiplication on a 4×4 cluster compared to the execution on a 2×2 cluster architectures).

The simulated instructions results are also provided per processor (Figure 4.15). These results are useful for the evaluation of an execution scenario. For a better understanding, only the results of a 2×2 cluster architecture (12 MicroBlazes and 1 ARM) are presented. It can be noticed for example that the ARM processor executes a greater number of instructions than the MicroBlazes in every matrix multiplication scenario. This is explained by the fact that the task running on the ARM processor is responsible for creating the matrices, splitting the computation, sending the computation data and collecting results from the MicroBlazes tasks. Moreover, it can be seen from the results that no MicroBlaze is blocked waiting for data, which means that there is no congestion on the network. Finally,

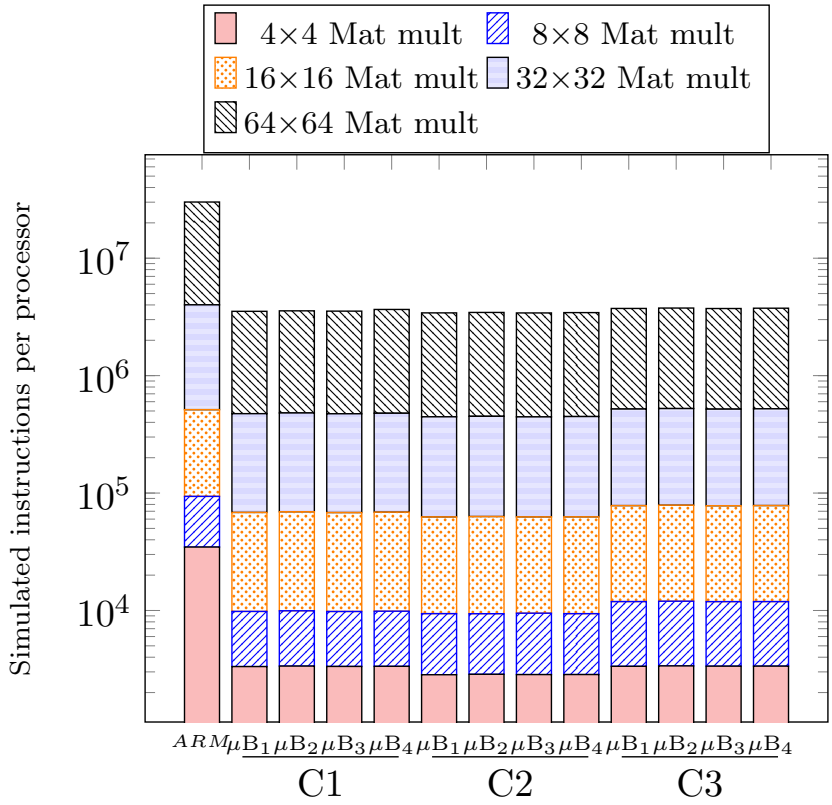


Figure 4.15 – Evaluation of the **simulated instructions per processor** for different sizes of matrix multiplications on a 2×2 architecture (12 MicroBlazes and 1 ARM)

in the evaluated scenario, the processors within the cluster *C3* in Figure 4.15, execute more instructions than processors within the rest of the clusters. This is because the cluster *C3* is the furthest one from the ARM, thus, it waits longer for the data sent by the ARM processor (memory polling instructions). The provided results could be also used in order to evaluate load balancing algorithms on such systems.

4.6 Conclusion

MPSoCSim is a simulator originally designed for the exploration of NoC design. The OVP technology has enabled the attachment of OVP processors and peripheral models to the existing SystemC simulator in order to be able to provide instruction accurate and very fast simulation of distributed memory MPSoCs running real application code. MPSoCSim

accuracy has been evaluated through the comparison with the hardware implementation. In this thesis work, this simulator has been extended in order to enable the modeling of complex clustered multi and many-core architectures with shared memory including private and shared resources. The execution results available with the extended version of this tool have been explored. The extended version of MPSoCSim meets the simulation requirements (execution time, flexibility, modeling and results capabilities) for the validation and comparison of the different deployment strategies proposed in Chapter 3. The next chapter presents and explains the comparison results using the extended version of MPSoCSim.

Chapter 5

Spatial isolation evaluation through virtual prototyping

Chapter contents

5.1 Experimental protocol 98

5.2 Cache attacks vulnerability case study 100

5.3 Deployment strategies comparison results 102

5.3.1 Considered deployment and execution scenarios 102

5.3.2 Results organization 103

5.3.3 Comparison according to each performance indicator 104

5.3.4 Comparison according to each execution scenario 108

5.3.5 Comparison summary 110

5.4 Conclusion 116

In this chapter, the extensions of the trusted manager in order to integrate the deployment strategies proposed in Chapter 3 are evaluated and compared through virtual prototyping using the extended version of MPSoCSim presented in Chapter 4. This chapter first explains the experimental protocol used for this evaluation. A case study demonstrating the cache attacks vulnerability of applications in the considered scenario is then presented. After, this chapter compares the different deployment strategies proposed in this thesis work, first, according to several performance indicators, and second, according to each considered execution scenario. Finally, this chapter draws some conclusions.

5.1 Experimental protocol

In this section, the experimental protocol used for the evaluation of the spatial isolation proposed in this work is presented.

This work has been evaluated through virtual prototyping using the extended version of MPSoCSim presented in Chapter 4. Besides the size of the NoC, this evaluation uses the same simulation parameters that those used for the validation of the first version of MPSoCSim in [90] summarized in Table 4.2.

For these explorations, a 4×4 cluster architecture is considered. This architecture is composed of 15 regular clusters encompassing 4 MicroBlazes each and one cluster encompassing one ARM processor. In total, this architecture offers 60 MicroBlazes and 1 ARM processors. The cluster's structure is shown in Chapter 4, Figure 4.6. Each processor has access by reading and writing to its own private memory as well as to a shared memory within its cluster. Additionally, each processor can access by writing to any other cluster memory. Shared memory is thus used for communication between processors (local and distant). In this work, as aforementioned, we consider a TSAR-like architecture. Clusters in this architecture include a (private) L1 cache per processor as well as a shared cache (L2) responsible of one segment of the shared memory. L2 caches are shared between all the processors in the architecture. In order to simulate this architecture on MPSoCSim, memories private to processors are seen as L1 caches and shared memories as shared L2 caches.

The trusted kernel services are executed on the ARM processor within a cluster dedicated to this purpose. This kernel is responsible for the dynamic management of the platform, for deploying applications as well as allocating, releasing and managing resources. It is responsible as well for the management of secure zones within which the isolated applications execute. Its services are explained in Chapter 3, Section 3.4. Finally, for simplicity reasons, in these experimentations we have considered non-preemptive scheduling. Consequently, each processor executes one task until its completion before another task is mapped and executed on the same processor. However, a preemptive scheduling kernel would not change the principle and evaluation results of the proposed approaches.

Furthermore, similarly to in Chapter 4, the same applications than used for the validation of the first version of MPSoCSim have been used for this exploration. Considered matrices size for these results is 64×64. Each application, is composed of 17 parallel tasks. Complete description of the applications is given in Chapter 4, Section 4.3.

A 17 parallel tasks application intended to be isolated allows the study of an unfavorable scenario for the considered system. Indeed, if isolated, the best case for the application in order to achieve its maximum parallelism is to execute within an isolated secure zone composed of all the resources it needs. In this case, this is 5 clusters. However, only 17 processors out of 20 will be used. In consequence, 3 out of 4 processors on the 5th cluster will not be used, which represents an unfavorable case for the considered architecture. Note that, when selecting a fixed secure zone size for its creation in static secure zone sizes approaches, both, the computing and the memory requirements, should be taken into account. Finally, for dynamic secure zone size approaches, a secure zone might be extended when either, computing or memory resources within the secure zone are not enough for the executing application. In these evaluations, we have considered the memory requirements only at the secure zone creation stage. Indeed, in these experimentations, only additional computing needs would require an extension of the secure zone. However, similar experimentation can be conducted considering dynamic memory requirements as well.

In this context, the kernel is required to know when each task is finished in order to declare available the corresponding processor. For this, each task sends a flag through the NoC to the kernel cluster. In this way, the kernel knows when a task or a complete application has finished.

Finally, in this context several independent applications are concurrently executing on the same platform. For this, the same application is duplicated in order to increase the load of the platform. For this experimentations, five applications corresponding to 86 concurrent tasks, are sufficient to compare the different application deployment strategies and execution scenarios. Each application includes a parameter indicating if the application is intended to be isolated. Finally, all the applications are supposed to be ready to execute from the beginning of the execution scenario. However, a priority level is associated to each application (master task) to determine in which order the ready tasks are served.

In the next section, these five applications are executed without any security mechanism in order to evaluate the cache attack vulnerability of each application on this considered execution scenario.

5.2 Cache attacks vulnerability case study

A study was conducted in order to evaluate the vulnerability to cache attacks of applications in a normal execution scenario on a TSAR-like many-core architecture with an average resource utilization rate of 77%. The kernel services are responsible for the dynamic deployment of applications. Tasks of each application may be spread across several clusters and thus may use and share several cluster memory banks (one L2 memory bank per cluster) with other applications. However, applications sharing cache memory with other applications (potentially malicious) are vulnerable to cache attacks. In these experiments, we first show, for each application (*Application* in Table 5.1), the number of clusters onto it is spread (C). Then, we highlight the number of memory banks (i.e., number of clusters) that each *Application* shares with other applications (M) as well as the identifiers of the different applications that it shares memory banks with (*Sharing application*). Finally, for each *Sharing application*, we measure the *Application* execution time (in percentage of its total execution time),*Sharing time*, that *Application* and *Sharing application* share each memory bank.

In fact, the cache attack vulnerability of each *Application* increases with M, Sharing time and the number of *Sharing applications*.

Experiment results are summarized in Table 5.1. As an example, let's consider Application 3. It is spread onto 5 different clusters. Consequently, it uses 5 different memory banks. Among them, 3 are shared with 3 different applications. Application 3 shares one memory bank with Application 2 during 83% of its execution time, one memory bank with Application 4 during 16.9% and one memory bank with Application 5 during 99.9% of its execution time.

From the presented results, it can be concluded that, in this scenario, where the average resources (computing and memory) utilization rate is 77%, resource sharing introduces an important cache attack vulnerability for each application (from 15.7% up to 99.9% of their execution time) that needs to be addressed.

To do so, we have proposed the spatial isolation of sensitive applications which prevents cache sharing for the isolated applications. In next section, the different application deployment strategies proposed in this work are evaluated.

Application	***C***	***M***	***Sharing time in %***	***Sharing application***
Application 1	5 clusters	1 memory bank	1 memory bank shared for 99.4% of the exec. time	Application 2
Application 2	5 clusters	2 memory banks	1 memory bank shared for 74.6% of the exec. time	Application 1
			1 memory bank shared for 99.9% of the exec. time	Application 3
Application 3	5 clusters	3 memory banks	1 memory bank shared for 83.0% of the exec. time	Application 2
			1 memory bank shared for 16.9% of the exec. time	Application 4
			1 memory bank shared for 99.9% of the exec. time	Application 5
Application 4	6 clusters	2 memory banks	1 memory banks shared for 75.6% of the exec. time	Application 3
			1 memory bank shared for 37.1% of the exec. time	Application 5
Application 5	8 clusters	2 memory banks	1 memory bank shared for 15.7% of the exec. time	Application 3
			1 memory bank shared for 45.6% of the exec. time	Application 4

Table 5.1 – Cache-based attacks vulnerability in the Baseline strategy where the average resources utilization rate is 77%

5.3 Deployment strategies comparison results

The ALMOS-like kernel services have been extended in order to integrate the spatial isolation of sensitive applications proposed in this work (Chapter 3). These extensions include different application deployment and resource management strategies. In this section the execution scenarios considered for these experimentations are first presented. Then, the results organization is introduced. Finally, results comparing the proposed deployment strategies according to both, different performance indicators and each execution scenario are discussed. A discussion about the comparison results concludes this section.

5.3.1 Considered deployment and execution scenarios

In this subsection, the considered deployment strategies and execution scenarios are explained.

Considered deployment strategies:

The deployment strategies explained in Chapter 3 have been evaluated and compared:

- **Baseline strategy** This scenario corresponds to the minimum kernel services before security-enable extensions. These services are presented in Chapter 3, Section 3.4. It does not include any security mechanism and no application is isolated.
- **Strategy A.1.** Secure zones with a fixed size composed of all the resources needed by the spatially isolated application in order to achieve its maximum parallelism (strategy explained in Chapter 3, Section 3.2.1). In the case considered in this work, the application parallelism is 17 tasks running in parallel and clusters are composed of 4 PEs. Consequently, in this case, the secure zone size is 5 clusters (20 dedicated PEs in total).
- **Strategy A.2.** Secure zones with a fixed restrained size limited to 4 clusters (4 instead of 5 in A.1. strategy) (see Chapter 3, Section 3.2.1).
- **Strategy B.1.** Fully dynamic approach (Chapter 3, Section 3.2.2).
- **Strategy B.2.** Dynamic approach with a guaranteed non-optimized secure zone size. In this case we fixed the guaranteed minimum secure zone size to 2 clusters (instead of 5 in A.1 strategy)(Chapter 3, Section 3.2.3).
- **Strategy B.3.** Resource reservation (Chapter 3, Section 3.2.3).

Considered execution scenarios:

For the evaluation of the performance overhead induced by each proposed strategy, different execution scenarios have been considered:

- **Scenario a.** Here, one single application (out of 5) is required to be spatially isolated within a secure zone. In this first scenario, this application has the highest priority (1 out of 5, 1 being the highest and 5 the lowest priority). Consequently, when mapped, there is no load on the platform. This is the best scenario for an isolated application.
- **Scenario b.** In this second scenario, the load of the platform is taken into account. In fact, as in scenario a, one single application needs to be isolated. However, unlike scenario a, this application has a medium priority (4 out of 5). This allows to take into account the load of the platform when trying to create the secure zone.
- **Scenario c.** In this last scenario, the load of the platform, as well as the number of applications requiring to be spatially isolated, are considered. In fact, here, 3 out of 5 applications, with priority levels of 1, 3 and 5 respectively, are required to be spatially isolated.

These execution scenarios have been considered and evaluated for each deployment strategy.

5.3.2 Results organization

The applications are first run concurrently without any secure-enable mechanism (Baseline strategy). Then, applications are concurrently run according to each couple of secure zone deployment strategy and execution scenario. Since approaches are deterministic, experiments are run once. Note that, execution scenario **a** is particular. In fact, in this scenario, the application intended to be spatially isolated is served the first one since it has the highest priority. Consequently, when the manager deploys it there is no load on the platform. As a result, strategies with a guaranteed minimum secure zone size (**B.2**) and with resource reservation (**B.3**) give similar results than 5 cluster static secure zone size strategy **A.1**. Indeed, in **B.2** strategy, only the kernel services execution time and total performance overhead are slightly different. Regarding **B.3**, it turns out to be identical than **A.1** since the algorithm finds a secure zone encompassing all the resources needed by the application (5 clusters secure zone) directly without requiring to reserve any cluster.

The main objective of these experimentations is to compare the different deployment strategies for the implementation of the spatial isolation in terms of induced performance overhead and to analyze them according to different performance indicators:

- *total execution time* for the set of applications where the kernel services execution time is included,
- average *execution time of isolated applications*,
- average *execution time of non-isolated applications*,
- average *resources utilization rate*,
- *Time spent on the trusted manager kernel services* impacted by the spatial isolation mechanisms,
- average time the isolated applications wait before being mapped (*waiting time*),

Apart from *resources utilization rates* in Table 5.2, the overhead on each performance indicator is always presented in terms of percentage compared to the Baseline strategy.

Results in Figures 5.1 to 5.5 and Table 5.2 allow the comparison of each performance indicator for each couple of secure zone deployment strategy and execution scenario (Section 5.3.3). Then, Figures 5.6 to 5.8 gather the results of every performance indicator classified by execution scenario (Section 5.3.4).

5.3.3 Comparison according to each performance indicator

The *total execution time* overhead for each couple of secure zone deployment strategy, and execution scenario, is presented in Figure 5.1. While the dynamism in the considered scenarios makes it difficult to explain every aspect of the results, several observations can be made. First, according to these results, the static 5 clusters Secure Zone (SZ) (**A.1** strategy) turns out to be the best solution in the evaluated scenarios providing the lowest overhead on the *total execution time*, almost negligible when there is no load on the platform (0.04%). It is interesting to notice that while limiting the size of the SZ to 4 clusters (**A.2**) entails a higher overhead than a 5 clusters SZ, the rest of the strategies do not seem to follow any trend but depend on the execution scenario. In order to better understand and compare these results, it is important to take into account that the *total execution time* is mostly impacted by two other performance indicators, the applications (isolated and non-isolated) execution time (*execution time of isolated and non-isolated applications*), as well as the *time spent on the trusted manager kernel services* for the

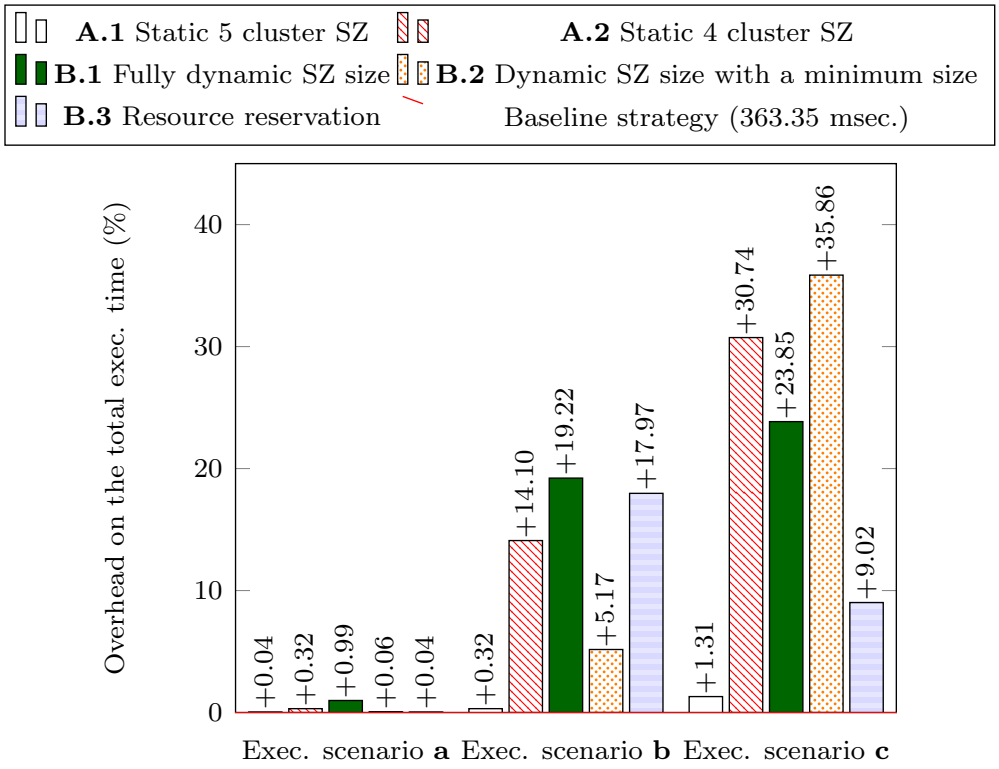


Figure 5.1 – *Total execution time* overhead (in percentage) compared to the Baseline strategy for each couple of secure zone deployment strategy and execution scenario

execution of the deployment mechanisms. Finally, note that applications and the trusted manager run in parallel.

In order to compare the impact of different deployment strategies, the *execution time of isolated applications* and *execution time of non-isolated applications* are highlighted in Figures 5.2 and 5.3 respectively. These results are presented in terms of induced overhead in percentage compared to the average applications execution time in the Baseline strategy. First, it can be seen that the static 5 cluster SZ (**A.1** strategy) always achieves a very good performance of isolated applications, but penalizes non-isolated applications. A better performance for isolated applications than in the baseline strategy is explained by the fact that an isolated application does not share its resources with any other application, and in this case it always achieves its maximum performance since the secure zone includes all the needed resources. This is not the case in the Baseline strategy, where the allocated

resources depend on the load of the platform. Moreover, as expected, when limiting the resources within the secure zone (**A.2** strategy) the performance of isolated applications is lower than in the first strategy. However, contrary to what it could be expected, limiting the secure zone size to 4 clusters (**A.2** strategy) instead of 5, further penalizes non-isolated execution time. This is explained by the fact that less resources are dedicated (4 clusters instead of 5) and thus not available for non-isolated applications, but for a longer time. Indeed, since the isolated application does not have all the needed resources, some of its tasks need to wait for available resources within the secure zone. Consequently, its execution time and the resources dedication time is longer. Furthermore, the fully dynamic strategy tends to penalize isolated over non-isolated applications. The dynamic size with a guaranteed minimum SZ (**B.2** strategy) size tends also to favor non-isolated over isolated applications performance. However, it guarantees a minimum number of resources for isolated applications achieving a better performance for isolated applications in cases where the platform load is very significant. Finally, the resource reservation strategy (**B.3** strategy) results vary to each execution scenario. Further performance indicators results presented below enable better understanding of this last strategy.

Moreover, *Time spent on the trusted manager kernel services* is illustrated in Figure 5.4. This is the time spent by the trusted manager for mapping, allocating and releasing resources (new application and task deployment, allocation of resources, creation and release of secure zones described in Chapter 3, Section 3.4.2). It is impacted by the deployment strategy and execution scenario. In Figure 5.4, the impact of each strategy on each execution scenario is presented in terms of induced overhead in percentage of the time spent on the Baseline strategy. It can be seen that dynamic strategies require a higher activity on the trusted manager compared to static secure zone size strategies (between 25 and 90% compared to 1.25 and 75% overhead for fully dynamic **B.1** and static 4 clusters **A.2** strategies respectively).

Furthermore, according to the secure zone deployment strategy and to the load of the platform, isolated applications need to wait for available resources before they can be deployed. In fact, in a static secure zone scenario, the trusted manager will wait until there are enough available resources before it can create a secure zone and deploy the isolated application. Figure 5.5 shows the average isolated applications *waiting time* before being deployed for each deployment strategy and execution scenario.

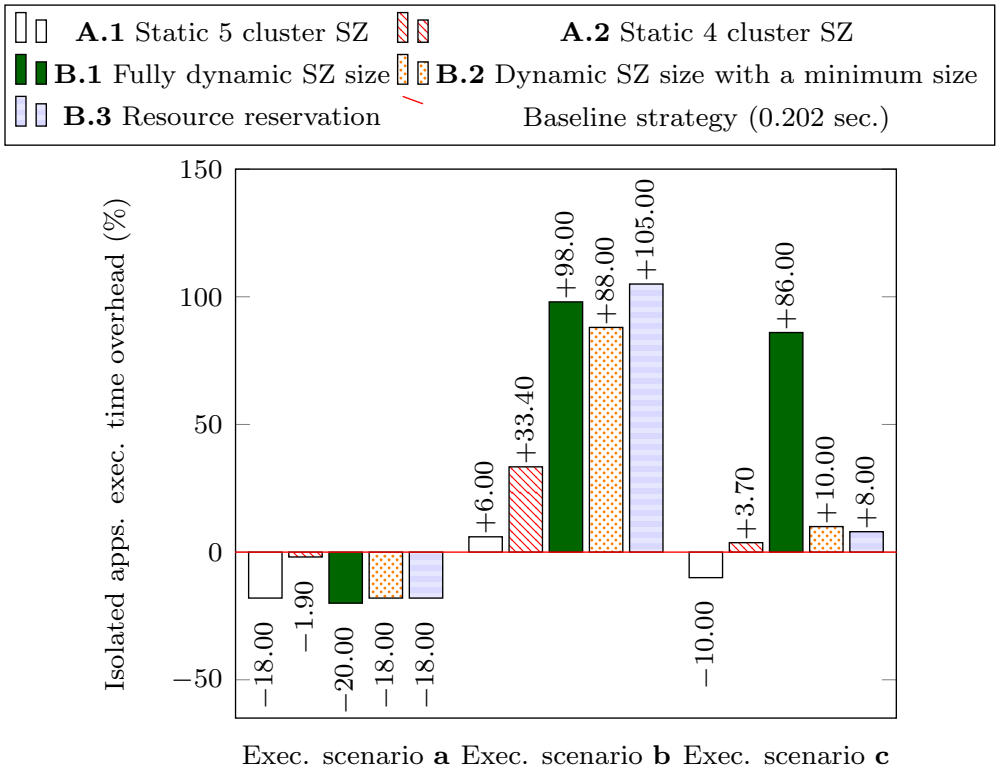


Figure 5.2 – Overhead on the average *execution time of isolated applications*, in terms of percentage of the average application execution time in the Baseline strategy for each couple of secure zone deployment strategy and execution scenario

Finally, in Table 5.2 the computing *resources utilization rate* is compared in total as well as within dedicated clusters to secure zones for each deployment strategy and execution scenario. While the resources utilization rate within secure zones allows the comparison between different deployment strategies with each other, the resources utilization rate in total shows the overhead of each deployment strategy compared to the baseline strategy (with a resources utilization of 77%). In this table, for each column the best and worst rate are highlighted in light and dark gray respectively. It can be noticed that since the fully dynamic strategy (**B.1** strategy) adapts the resources the best to the needs of applications and load of the platform, it achieves the best resources utilization rates.

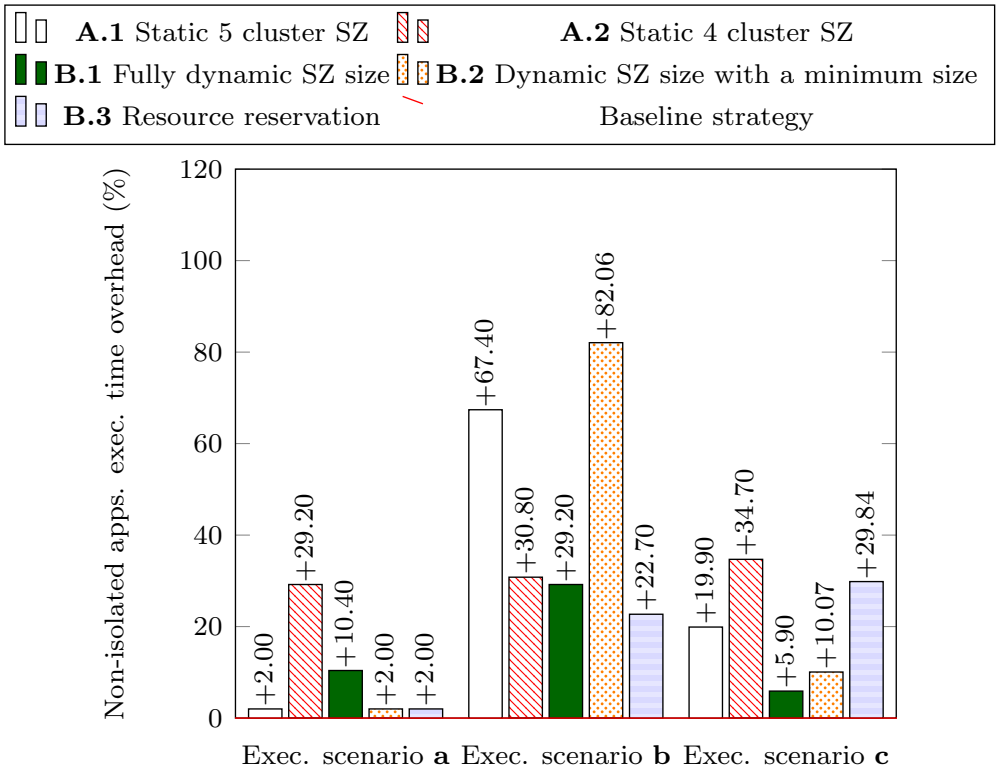


Figure 5.3 – Overhead on the average *execution time of non-isolated applications*, in terms of percentage of the average application execution time in the Baseline strategy for each couple of secure zone deployment strategy and execution scenario

5.3.4 Comparison according to each execution scenario

Figures 5.6, 5.7 and 5.8 gather the results presented above in order to allow the comparison of the proposed strategies according to every studied performance indicator for each execution scenario (scenario a, b and c in Section 5.3.1). Each result is presented in terms of induced overhead in percentage of the corresponding performance indicator in the Baseline strategy (dashed line at 0% on the chart). Moreover, the scale is arranged. In fact, results are presented in such a way that, for each performance indicator value, the closest to the chart border, the better. Consequently, the best strategy for each performance indicator is the closest one to the chart border. Finally, for readability reasons, results concerning the *waiting time* are presented in their log value.

The objective of these charts is to provide a quick overview of the interest of each

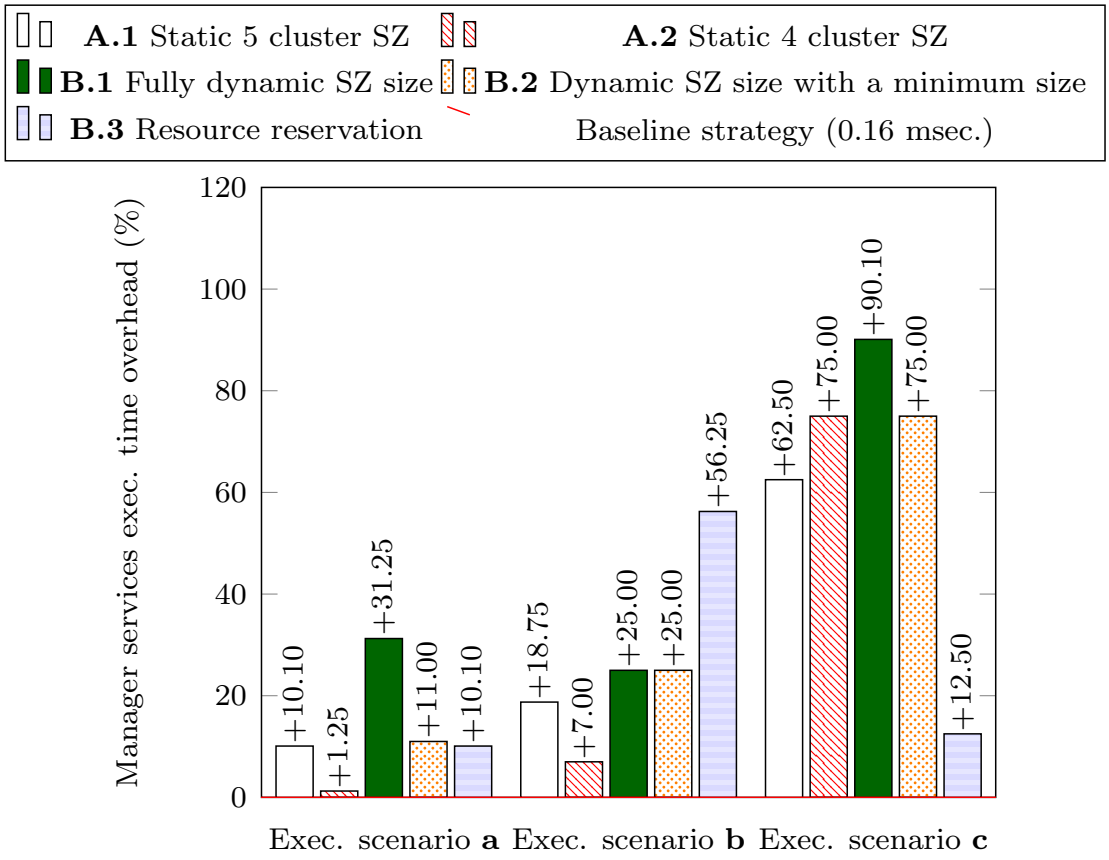


Figure 5.4 – *Time spent on the trusted manager kernel services* impacted by the secure-enable mechanisms in terms of percentage compared to the Baseline strategy for each couple of secure zone deployment strategy and execution scenario

deployment strategy in each execution scenario, in contrast with results in last subsection which provide precise values.

From these charts it can be seen that, the *best* deployment strategy in each execution scenario depends on the performance indicators to favor. Consequently, depending on this, one can decide, by choosing the deployment strategy, which performance indicators will absorb the performance overhead induced by the spatial isolation mechanisms. This provides flexibility on the control and management of the platform.

Taking into account the presented results, next subsection compares the considered deployment strategies.

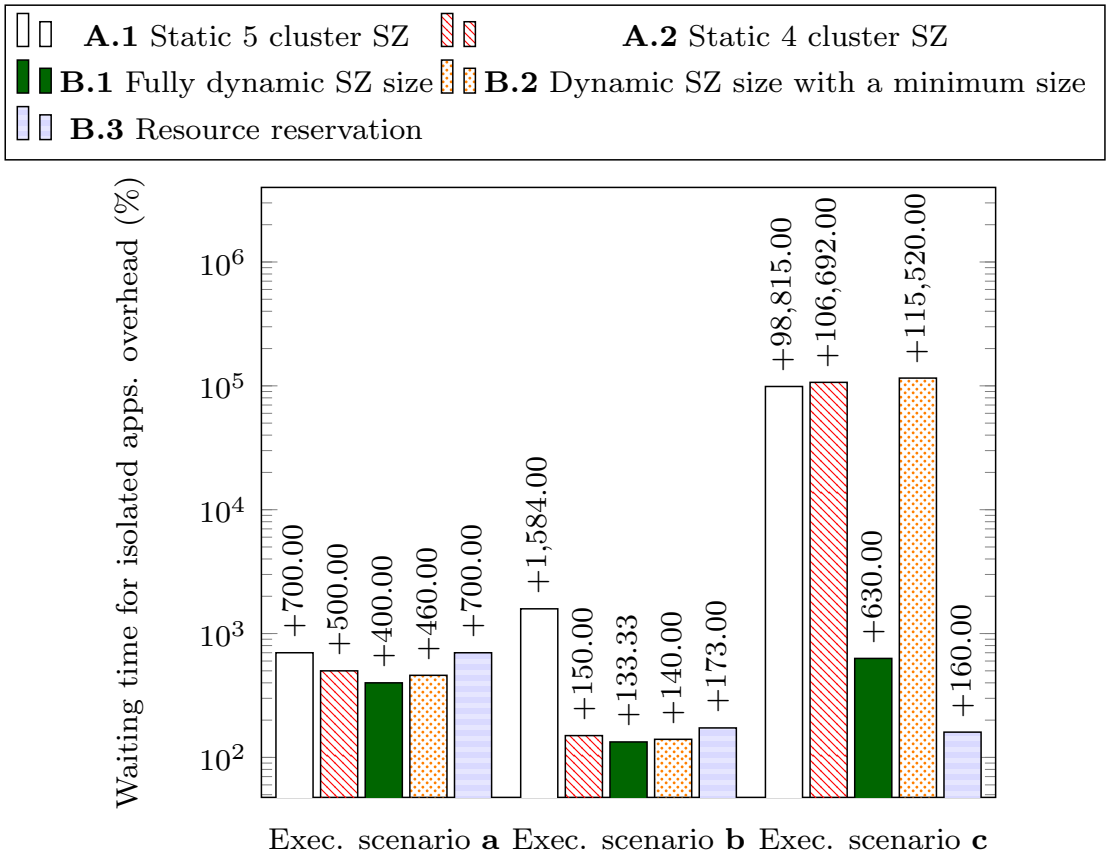


Figure 5.5 – Overhead on the average *waiting time* before the deployment of isolated applications, in terms of percentage compared to the Baseline strategy, for each couple of secure zone deployment strategy and execution scenario

5.3.5 Comparison summary

- Static approach: When the size of the secure zone is well chosen or includes all the resources needed by the application, this approach is the best solution for the performance of isolated applications. However, when the secure zone size is not well chosen, for example in the **A.2** strategy, applications performance (both isolated and non-isolated) may be very penalized. In fact, in **A.2** strategy, only 4 clusters instead of 5 are dedicated to the isolated application. Unlike what could be expected, this solution further penalizes non-isolated applications than in **A.1** because the resources are dedicated much longer since some isolated application tasks need to wait for

	Exec. scenario a		Exec. scenario b		Exec. scenario c	
	SZ	**Total**	**SZ**	**Total**	**SZ**	**Total**
Static approach, optimal size (5 clusters)	85%	68,5%	85%	71%	85%	61,6%
Static approach, limited size (4 clusters)	65%	64%	65%	69%	65%	55%
Fully dynamic	85%	72%	89%	69%	92%	67%
Dynamic approach with guaranteed minimum SZ size	85%	68.5%	81%	60%	85%	55,6%
Resource reservation	85%	68.5%	85,4%	67%	86,2%	65%

Table 5.2 – *Resources utilization rate* within secure zone(s) as well as in total (referred to as *Secure zone* and *Total* in the table) for each couple of secure zone deployment strategy and execution scenario. For each *SZ* and *Total* columns, the best and worst resources utilization rates are highlighted in light gray and dark gray respectively. The *resources utilization rate* in the Baseline strategy is 77%

other tasks within the same secure zone to release their resources. Consequently, the isolated application execution time is much longer. On the other hand, in static secure zone size approaches, isolated applications need to wait longer to start their execution depending on the availability of resources. However, once they are mapped, they may achieve very good performance. In conclusion, this approach is the most interesting when the performance of isolated applications is a priority, however it requires a good knowledge of the isolated applications in order to choose a good size of the secure zone.

- Fully dynamic approach: This approach is the best one when it is required to maximize the resources utilization rate within the secure zones as well as at the platform level. Also, this strategy is suitable when the performance of isolated applications is not a priority or when the isolated application is not known, making it impossible to choose a good secure zone size. Indeed, this approach does not entail a significant performance overhead on the total execution time (*total execution time* overhead from 0.99% in execution scenario **a**, up to 23.85% in scenario **c**). However, this approach tends to penalize the performance of isolated applications and entails a high activity on the trusted manager services due to secure zone size dynamism.
- Dynamic approach with a guaranteed non-optimized secure zone size: This approach is a good trade-off between fully dynamic and completely static approaches. Indeed,

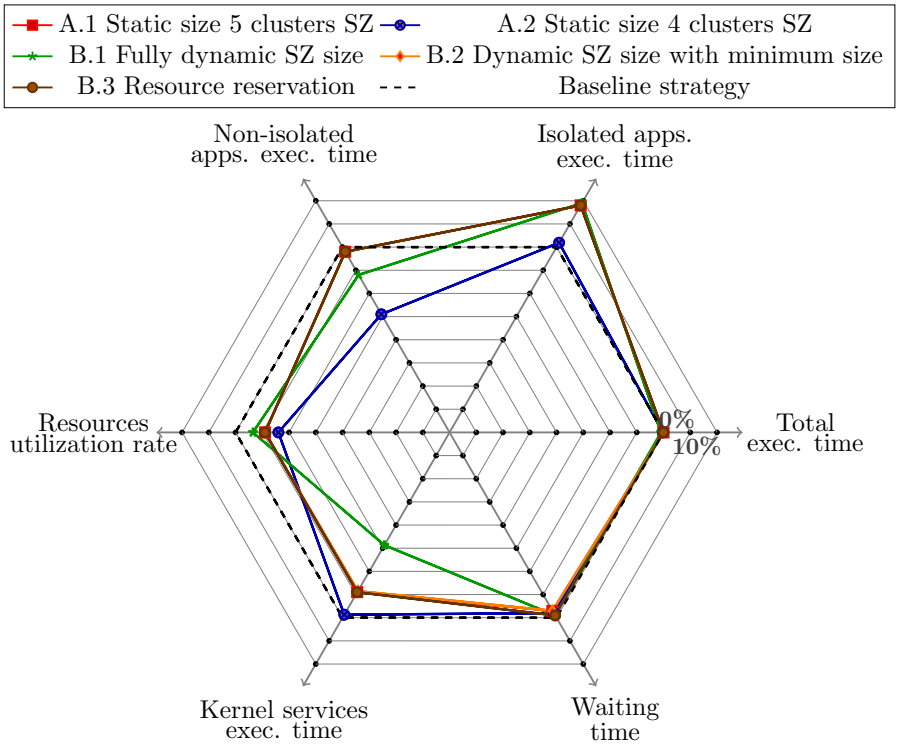


Figure 5.6 – Comparison of different strategies - Exec. scenario **a**. Values are presented in terms of induced overhead in percentage with respect of the baseline value. **Arranged scale**: the closest to the chart, the better. Scale: 1 division : 10%

it guarantees a minimum number of dedicated resources to the isolated application, thus a minimum performance level. Also, this approach takes into account the load of the platform when trying to extend the secure zones. As a result, non-isolated applications result less penalized than in **A.1** strategy. Moreover, due to its dynamic side, this approach achieves a good resources utilization rate, but entails more activity on the trusted manager compared to a static approach. Finally, it requires having some knowledge on the application meant to be isolated in order to choose the minimum required secure zone size.

- Resource reservation: Resource reservation is an interesting approach but depends on the selection of resources to reserve. Consequently, more sophisticated metrics are required in order to increase the chances of extending a secure zone and achieving good performance of isolated applications. In fact, the size of the secure zone as

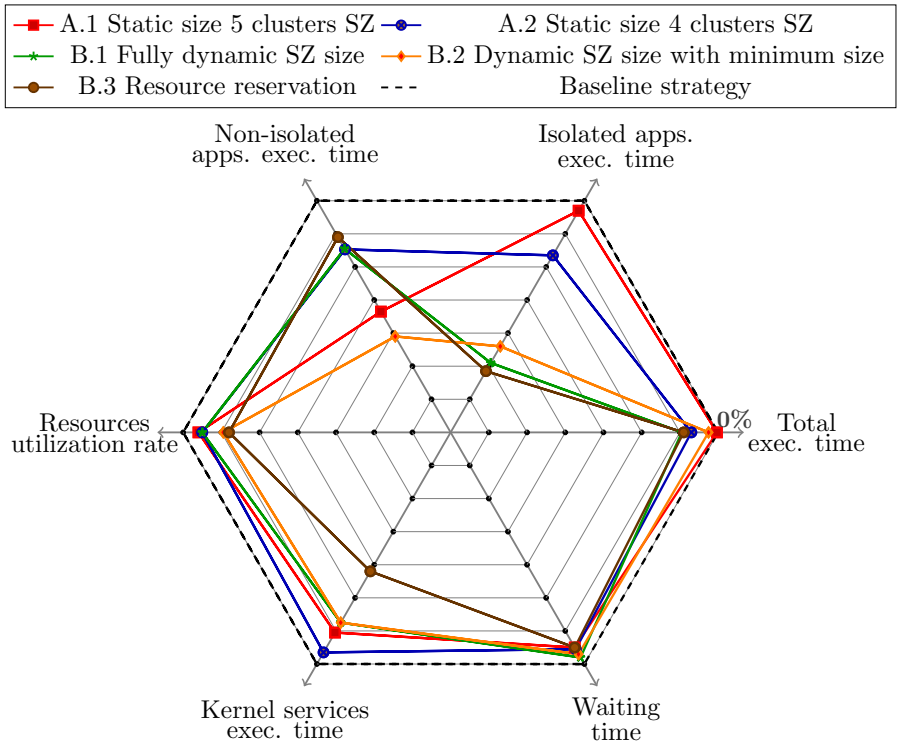


Figure 5.7 – Comparison of different strategies - Exec. scenario **b**. Values are presented in terms of induced overhead in percentage with respect of the baseline value. **Arranged scale**: the closest to the chart, the better. Scale: 1 division : 20%

well as the performance of the isolated application totally depend on the load of the platform when creating a secure zone as well as when selecting the resources to reserve. On the other hand, this will lead to an increased complexity of the kernel manager algorithms. Moreover, in this approach, there is no guarantee of performance.

Finally, it is interesting to notice that in this work, application migration is not considered due to the induced complexity and cost. However, application migration could be useful in two ways. First, migration could be used together with the dynamic secure zone size approaches discussed in this work in order to dynamically migrate a secure zone when a larger number of contiguous clusters is available. Second, migration of running non-isolated application would be useful in order to rearrange the execution on the platform gathering the available resources in order to maximize the chances to find a bigger secure

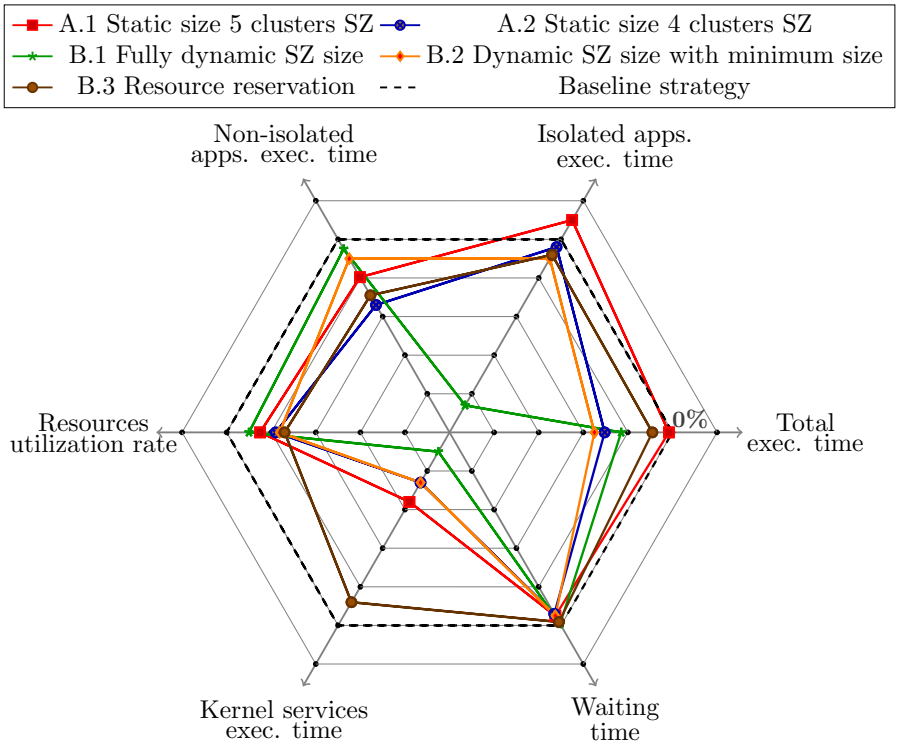


Figure 5.8 – Comparison of different strategies - Exec. scenario **c**. Values are presented in terms of induced overhead in percentage with respect of the baseline value. **Arranged scale**: the closest to the chart, the better. Scale: 1 division : 20%

zone and being able to create a fixed secure zone size more quickly. In terms of security, the second approach which considers migrating non-isolated applications would be more suitable since it would not require secure migration of secure zones applications and data.

Table 5.3 shows for different execution scenario characteristics, which studied deployment strategy would be particularly suitable (check mark), not particularly suitable (dash) and not suitable (cross mark). Consider for example the execution scenario in which the application meant to be isolated is known. In this case, all the strategies except the fully dynamic one, take advantage of this scenario. Indeed, the application resources requirements are necessary to select a suitable size of secure zone, or a suitable number of reserved resources. On the other hand, the fully dynamic strategy does not require any knowledge of the application to be isolated.

Execution scenario	Static SZ size		Dynamic SZ size		
	Optimal size	**Limited size**	**Fully dynamic**	**Dynamic with guaranteed size**	**Resource reservation**
Isolated applications are known	✓	✓	-	✓	✓
Isolated application's performance is critical or is required to be guaranteed	✓	✓	✗	✓	✗
Non-isolated application's performance need to be favored	✗	-	✓	✓	✗
Resources utilization rates need to be favored	-	-	✓	-	-
Low load on the platform	✓	✗	-	-	-
Migration is available	✓	✓	✓	✓	✓
Kernel services activity need to be minimized	-	-	✗	-	-

Table 5.3 – Selecting the deployment strategy summary

5.4 Conclusion

In this thesis work, the spatial isolation for sensitive applications against access-based cache SCAs is proposed. For its implementation, the services of the OS kernel are extended in order to integrate different application and resource allocation strategies proposed in this work. MPSoCSim has been extended in order to be able to compare the implemented strategies on a TSAR-like many-core architecture. In this chapter, different results comparing the proposed strategies are presented. Different scenarios and capabilities of each strategy are discussed. While static size secure zone strategies are interesting when isolated applications performance is critical, these strategies require to have some knowledge of the isolated application in terms of resources and penalize non-isolated applications performance. On the other hand, if performance of the isolated applications is not critical, then dynamic secure zone size strategies are more suitable since they take into account the current load of the platform and achieve better resource utilization rates. However, these approaches require higher activity on the kernel services and do not guarantee any performance level. Finally, trade-off approaches are interesting, achieving, without any performance guarantee, a better balance between isolated and non-isolated performance overheads.

Chapter 6

Conclusion and future work

Chapter contents

6.1 Summary 117
6.2 Spatial isolation discussion 118
6.2.1 Back to the state-of-the-art 118
6.2.2 Further attacks and spatial isolation capabilities 120
6.3 Possible improvements and leads for future work 121
6.4 Conclusion 128

In this chapter, the thesis work presented in this manuscript is first summarized. Then, the spatial isolation proposed in this work is discussed. For this, the positioning of the proposed solution compared to most similar state-of-the-art work is first given. Then, some capabilities of this work regarding attacks not considered in our threat model are presented. Finally, possible improvements and future work are discussed.

6.1 Summary

This thesis work considers the problem of cache-based SCAs on multi and many-core architectures. For this purpose, different contributions have been presented in this thesis manuscript on access-based cache SCAs countermeasures and their evaluation, as well as on virtual prototyping tools for simulation of clustered multi and many-core systems.

This thesis work proposes the spatial isolation of sensitive applications in order to address the problem of cache-based attacks. This solution relies on the fact that cache-based attacks are introduced by cache sharing between sensitive and malicious applications, which

allows these latter to monitor their own performance in order to deduce the victim process' cache activity. Based on this statement, the solution proposed in this work aims at preventing a sensitive application from cache sharing by executing it within a dedicated spatially isolated environment called secure zone. A secure zone is composed of a number of contiguous clusters that are temporarily dedicated to a sensitive application. Thanks to the dedication of these resources, the cache activity of the isolated application cannot longer be monitored and, as a result, access-based cache SCAs cannot longer be performed against the isolated application. In order to implement this solution, we have proposed the extension of the OS kernel services in order to integrate the mechanisms responsible for the dynamic deployment of applications and management of resources. For this, several applications deployment and resources management strategies have been proposed. For their evaluation and comparison, MPSoCSim, an OVP-based simulator has been extended. Results generated through the extended version of MPSoCSim show that the spatial isolation is a flexible solution that allows to manage the performance indicator that is impacted by the spatial isolation mechanisms, according to the chosen deployment strategy. Consequently, the deployment strategies can be selected according to the user requirements, considered execution scenario and load of the platform.

6.2 Spatial isolation discussion

In this section, the spatial isolation proposed in this work is discussed according to similar state-of-the-art solutions as well as according to further attacks not considered in our threat model.

6.2.1 Back to the state-of-the-art

In this work, the spatial isolation of sensitive applications has been proposed for multi and many-core architectures against access-based SCAs. Each sensitive application executes spatially isolated within a secure zone. Resources within the secure zone are not allocated nor shared with any other application. Since, caches within the secure zone are not shared with any other application, the isolated application's cache activity cannot longer be monitored. Consequently access-based cache SCAs cannot longer be performed against the isolated application. This solution relies on the fact that the caches used by the isolated application are not used by any other application.

Similarly, other state-of-the-art countermeasures aim as well at avoiding cache sharing. These are explained and discussed in Chapter 2. However, they propose solutions at a different granularity and require different mechanisms. Cache coloring [70] for example, proposes to statically or dynamically partition the cache and allocate to sensitive applications different partitions than those allocated to other applications. The granularity of this solution is then a cache partition, usually composed of a certain number of cache sets. However, these solutions require hardware modifications on the micro architecture in order to partition and color the cache. Moreover, page locking [69] addresses the fact that a key cause of cache attacks is the cache interference when a cache is shared between sensitive and potentially malicious applications. Their objective then, is to prevent cache interferences. Interferences are caused by the eviction of memory pages from the cache, when the memory pages belong to the sensitive application, but they are evicted by accesses of a different (potentially malicious) application. Based on this, this solution proposes to allocate some memory pages to the sensitive application that, once in the cache, cannot be evicted by other applications during the entire execution time of the sensitive application. This approach provides a solution at the granularity of memory pages. However, it requires each application to be able to know which information is sensitive in order to store it on locked memory pages.

In contrast with these approaches, the solution proposed in this work takes advantage of the large number of resources offered by many-core architectures as well as of some memory features of available many-core architectures. For this work, we consider a TSAR-like architecture that provides logically shared but physically distributed memory. Indeed, memory is partitioned into segments from the architecture design and each segment is mapped to a LLC on a cluster (Chapter 1, Section 1.2.1). This feature, originally for data locality and performance purposes, makes easier the implementation of the spatial isolation mechanisms. In fact, this is a trend in many-core architectures since centralized memory does not scale in large architectures. Moreover, spatial isolation in this context, provides a solution at the system level and does not require hardware mechanisms. In fact, here the problem is treated as a deployment and resource allocation problem. Its implementation, at the OS kernel level, offers a flexible solution, where the applications deployment and resource management can be chosen according to the user and applications requirements as well as according to the load of the platform.

6.2.2 Further attacks and spatial isolation capabilities

NoC attacks: According to the threat model considered in this work (presented in Chapter 1, Section 1.2.2), the communication through the NoC is considered secured. In fact, in this work, all the resources within the secure zones are dedicated to one single application except for the NoC. Consequently, the NoC is still shared between all the applications. In literature, there are some works that consider the problem of information leaked through the NoC communication [100][101]. However, little work can be found on practical attack implementations. In [37], authors propose a timing attack on NoC using PRIME+PROBE technique [29]. However, authors consider MPSoC platforms where several processors (each on a different node) share a distant LLC. Based on this, the NoC traffic when accessing to the distant shared LLC can be monitored and exploited by an attacker. Consequently, this attack relies on the fact that there is only one LLC and that it is shared between all the processors on the platform. This is not the case in many-core for scalability reasons. In fact, a cache on a certain node being shared between a great number of distant processors would cause a memory bottleneck. In the system considered in this work, the LLC has special features. Indeed, the LLC is distributed among the clusters which makes very difficult to know, from the NoC traffic, which application is accessing to which LLC (note that it would be interesting to see the capabilities of a distributed attack in terms of several distant malicious tasks working together in this scenario). This information is only available within a cluster (through bus communication, not through NoC), where there is one part of the LLC directly shared on the bus between the processors on the same cluster. In order to counter this, this thesis work proposes the spatial isolation preventing one sensitive application from sharing clusters (caches and buses among others) with other applications.

To our best knowledge, there is no attack proving their practicality on many-core architectures including distributed LLCs. However, some literature countermeasures addressing these NoC attacks are compatible with our work. For instance, some generic NoC countermeasures have been proposed in [73] (See Chapter 2, Section 2.1.2 for further explanation). Here, authors propose to disturb the possible attacker observations on the NoC traffic by some different mechanisms. One of the proposed mechanisms is to adopt a non-deterministic routing protocol instead of the traditional deterministic XY protocol in order to add non expected behavior on the NoC communication that can disturb the attacker measurements. In [73], authors use a semi-adaptive west-first routing

logic. Fully adaptive routing policies could improve the chances to disturb the attacker. In the simulation environment used for the work presented in this manuscript, XY as well as semi-adaptive west-first routing logic are supported among others. West-first routing logic can thus be used together with the spatial isolation mechanisms proposed in this work. These two solutions have been evaluated together in our environment (for the deployment strategy **A.1** and execution scenario **a** described in Chapter 5, Section 5.3.1) showing a deviation on the total execution time of 2% on the considered scenario.

Confidentiality attacks by illegal direct access to data: In this work, we have addressed cache-based SCAs. These attacks allow an attacker to indirectly accessing secret information by observing and exploiting side-channel information, in this case time variations. It is interesting to notice that, as mentioned in Chapter1, Section 1.2.4, other shared hardware components such as the TLB and BTB leak information due to thread contention. While not specifically addressed, the spatial isolation mechanisms proposed in this thesis work could protect sensitive applications from these attacks since it prevents the sharing of these physical resources.

On the other hand, in this work we have not considered the illegal direct access to data (see threat model described in Chapter 1, Section 1.2.2). In fact, this work considers the use of other mechanisms specifically addressing these attacks such as MMU and/or MPU.

DoS attacks: Finally, it is interesting to notice that while DoS are not considered in our threat model (Chapter 1, Section 1.2.2), the spatial isolation proposed in this work, prevents as well DoS attacks on the resources within the secure zones (processing, memory and bus communication). Indeed, the trusted kernel is responsible for the allocation of resources according to whether the application is intended to be spatially isolated. Thanks to the dedication of resources within the secure zones clusters, any other application is able to use these resources nor to launch DoS on these resources.

6.3 Possible improvements and leads for future work

In this section, some possible improvements and future work leads are presented.

Integration of spatial isolation mechanisms on SoCLib cycle-accurate, bit-accurate implementation of TSAR-ALMOS system: The spatial isolation mecha-

nisms have been evaluated in Section 5 through the extended version of the OVP-based MPSoCSim on a TSAR-like architecture. After this exploration, the most interesting deployment strategies can be evaluated on the SoCLib implementation of the ALMOS-TSAR system. The extension of the ALMOS OS in order to integrate the spatial mechanisms is currently being addressed. The objective is then to extend the ALMOS services in order to be able to declare a secure zone and to allocate resources and deploy applications on TSAR accordingly. For this preliminary work, we consider a single secure zone (i.e., only one sensitive application requiring to be spatially isolated). Moreover, for this first part of the work, the secure zone is declared at the architecture design time.

Once, the secure zone is declared in the TSAR architecture, four main ALMOS services are extended: the monitoring, the application and mapping allocation and memory allocation services.

First, the secure zone is declared at the design time on the TSAR architecture. The TSAR hardware specifications are described in a Binary Information Block (BIB) format in order to pass this information to *info2bib* utility for the generation of the binary format. The file *.info* used on this generation has been modified in order to include an additional *Secure* parameter to each cluster (i.e., *cluster_s* structure). This parameter indicates whether a cluster is dedicated to a secure zone.

Second, in order to take into account the new parameter within ALMOS, the ALMOS monitoring structure (DQDT) required to be extended at three stages: its creation, its initialization and the propagation of the architecture parameters values (i.e., through the structures *struct boot_info_s* and *arch_bib_cluster_s*).

Third, each thread requires an additional parameter indicating whether it belongs to the secure application. The value of this parameter is set by the user. This information will be taken into account by ALMOS when deploying it and allocating resources. A new attribute *Secure* has been added to the POSIX threads attributes considered by ALMOS. Furthermore, the necessary functions in order to set and read the new thread parameter value have been added.

Finally, this new value is taken into account by consulting the monitoring structure when mapping a new thread (i.e., *dqdt_do_placement*), a new application (*sys_thread_create*) as well as when allocating memory (*dqdt_mem_do_request*).

This work is in progress and its evaluation is currently being addressed.

Considering different applications: First, in this work we have focused on the

comparison of the performance overhead induced by different deployment and resource allocation strategies. In order to make sure that only the strategies impact the performance overhead measurements, we have considered the same application duplicated a certain number of times in order to increase the load of the platform. For these experimentations, matrices multiplication has been considered. Using other applications and comparing the obtained results would improve this work.

Supporting migration: In this work, application migration is not considered due to the induced complexity and cost (Chapter 1, Section 1.2.2). Indeed, migration here would include the secure remapping of the application and processor context switch as well as the memory remapping of the application data and instructions in order to leverage data locality. However, supporting applications and secure zones migration would improve several aspects of this work.

First, the deployment of concurrent secure zones can cause the fragmentation of non-secure clusters into non-adjacent partitions. This may increase the spatial distance between non-secure clusters and thus the communication cost between non-isolated application tasks. Migration of non-isolated and/or secure zones migration can address fragmentation. In fact, migrating applications would allow the rearrangement of available and dedicated resources in order to gather secure zones and in this way optimize the data locality and performance of non-isolated applications. However, for dynamic size secure zones strategies, gathering secure zones could prevent them to find available contiguous resources in order to be extended. It would be interesting to consider migration in order to explore this point.

On the other hand, allowing migration for non-isolated applications only (not secure zones) could favor the creation and extension of a secure zone. Indeed, spatially gathering non-secure applications, would result in larger available partitions (available contiguous clusters) favoring the creation and extension of secure zones.

Finally, application migration might be considered in the future in order to cope with problematics such as dark silicon, component aging, faulty components, etc.

Optimizing the proposed deployment strategies and further explorations: Several optimizations on the proposed deployment strategies are possible. First, resource reservation strategy (Chapter 3, Section 3.2.3) can be improved. In fact, the selection of the clusters to be reserved can be done according to smarter metrics in order to favor the possibility of extension for secure zones. For example, the current load of the platform can

be taken into account when selecting clusters to reserve.

Moreover, in the presented experimentation work, only one deployment strategy is used per execution scenario in order to be able to analyze the comparison of other factors such as load of the platform and number of concurrent secure zones (Chapter 5, Section 5.3.1). However, in a further study, it will be interesting to consider dynamically selecting a deployment strategy for a secure zone that can be different from the one used for another concurrent secure zone. The strategy to use could be selected according to the current load of the platform, or to the isolated application requirements. For example, let's consider an execution scenario in which the load of the platform is very low. In this case, if an application intended to be isolated is ready to be deployed, then, a static size secure zone including all the resources that the application needs in order to achieve its maximum parallelism (see chapter 3, Section 3.2.1) can be used. Later on, if the load of the platform has increased and an application intended to be isolated is ready to be deployed but its performance is not critical, then, a different deployment strategy that does not guarantee/favor the isolated application performance can be used, for instance the fully dynamic size secure zone (Chapter 3, Section 3.2.2). Dynamically selecting the deployment strategy of each secure zone can favor the good management and utilization of resources.

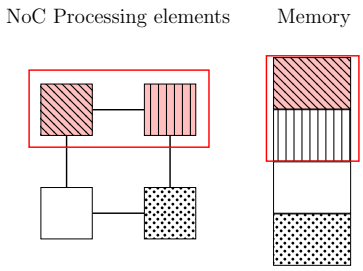
Furthermore, it could be interesting to consider non-contiguous secure zones, which would give more flexibility when creating and extending secure zones. However, the isolated applications being spread onto distant clusters, their communication cost would increase. Moreover, non-isolated applications performance would be impacted as well. The interest of this approach compared to contiguous secure zones is interesting to study in terms of induced overhead on both, isolated and non-isolated applications.

Considering the communication between applications: In this study, independent applications are considered. As a consequence, the communication between applications has not been addressed and might be studied in future work. In order to secure sensitive applications communication against any possible information leakage, a possible solution would be to have two different NoC communication channels *normal* and *secure*, according to each application security requirements. Applications would require to authenticate themselves in order to be able to use secure channels. Moreover, a segment of memory shared for communication could be considered. In this case, when an application sends information to a second application, the data is stored in the declared shared memory. The second application can then retrieve the information and store it in

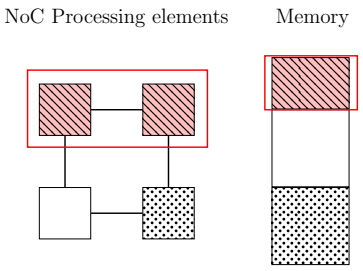
its local memory (e.g., memory within its secure zone).

Dynamic memory-to-cache mapping on TSAR architecture: The dynamic memory-to-cache mapping is currently being studied in order to reduce the under-utilization of resources that can be induced by the spatial isolation proposed in this work, in the TSAR architecture. In fact, in the spatial isolation approach, all the resources of clusters within a secure zone, memory and processing resources, are dedicated to the isolated application. In this work, we consider a TSAR-like architecture. Section 1.2.1) in Chapter 1, explains the specificities of this architecture. One of those is the static segmentation of the shared memory. The number of segments is equal to the number of clusters, and each segment is mapped to a cluster LLC. Dedicating a cluster to a secure zone entails the dedication of the processing resources, as well as the dedication of the corresponding memory segment. Consequently, two worst scenarios in terms of utilization of resources are possible. An entire cluster could be dedicated to an isolated application secure zone when the application needs the processing elements of the cluster only (see under-utilization of memory in Case 1 in Figure 6.1), or when the application needs memory on an extra memory segment but no further processing elements (under-utilization of processing elements in Case 2 in Figure 6.1).

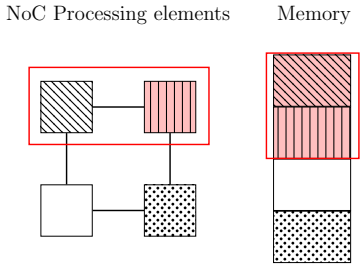
In order to cope with the potential under-utilization induced by these mechanisms, the dynamic memory-to-cache mapping can be explored. Let's consider Figure 6.1. On the left size of the figure one of the two cases aforementioned is illustrated, while on the right side, a dynamic memory-to-cache mapping solution is proposed. Each of the four figures illustrated is composed of a NoC and a memory part. The NoC on the left is composed of four clusters. Each cluster is composed of some processing elements, each with a L1 cache, and a shared LLC between the processing elements within the clusters. This is further described in Chapter 1, Section 1.2.1. On the right, the segmented memory is illustrated. Each memory segment is mapped on a specific cluster. In the figure, each memory segment has the same pattern (vertical, diagonal lines, none or dots) that the cluster it is mapped to. Each of the four figures presents a secure zone (red rectangle) both, on the NoC as on the memory sides. Red-filled secure zone memory segments indicate segments that the isolated application does use. While white-filled ones indicate that while the segments are dedicated to the secure zone, they are not actually used. On the NoC side, for understanding purposes, the red-filled clusters give indication on the processing resources only. These indicate clusters within the processing resources that are used, while



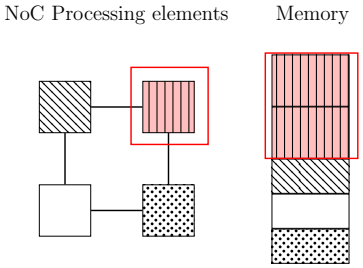
Case 1: Under utilization of memory



Solution to case 1: memory-to-cache remapping



Case 2: Under utilization of processing elements



Solution to case 2: memory-to-cache remapping

Figure 6.1 – Dynamic memory-to-cache mapping to reduce under-utilization or resources

the white ones indicate those that are not used. Finally, for simplicity reasons, in the rest of this subsection, the word *cache* will refer to LLC.

In Case 1, the Figure 6.1 shows an isolated application using the processing elements of two clusters but using in terms of memory size, only one memory segment. Consequently, the memory is under utilized. The dynamic mapping of memory to caches would made possible the remapping of the non utilized memory segment onto other(s) cache(s). Consequently, the memory under-utilization in this case could be prevented. In Case 2, the opposite scenario is illustrated. Here, an isolated application uses two memory segments but the processing elements of only one cluster. However, two entire clusters are dedicated to its secure zone. A possible solution in order to prevent the processing elements under-utilization in this case would be to remap the two used memory segments to the cache which processing elements are utilized by the application.

However, this approach would impact, in a different way according to the scenario, the performance of both, isolated and non-isolated applications. In fact, different scenarios are conceivable. In Solution to Case 1 illustrated in Figure 6.1 for instance, the isolated application executes onto two clusters, consequently it can use two caches. However, the application uses only one segment of the memory which is remapped onto the two caches. As a consequence, the number of cache misses for this isolated application might be reduced (one memory segment for twice the size of the cache). On the contrary, the rest of the segments are remapped onto the rest of the clusters caches. Note that this remapping could be done equally onto the rest of the caches or in any other fashion. After this remapping, the rest of the caches would be responsible for a larger memory segment which will potentially entail a large number of misses, impacting the performance of non-isolated applications using these clusters. On the contrary, in Solution to Case 2 in the figure, two memory segments used by the isolated application are remapped to one single cache, this would entail a larger number of cache misses. However, the data locality would be improved, since every data access is local (onto the same cluster). On the other hand, the rest of the memory segments (in this case two) are remapped onto the rest of the caches (in this case three). Consequently, the non-secure caches are responsible for a smaller segment which would entail a smaller number of cache misses. However, non-isolated applications data might be spread onto a larger number of clusters, due to their smaller size, which would penalize the data locality and potentially increase their communication cost.

As explained here with these two examples, different scenarios are conceivable, and for each, different dynamic parameters could impact isolated and non-isolated applications in a different way. It would be thus interesting to further explore and study the impact of this kind of solution.

Concerning the implementation of the segmentation and dynamic mapping of memory to cache, there are several possibilities. For instance, the memory mapping to cache could be done entirely dynamically according to the needs of applications (entire rearrangement of memory). However, this would be very complex and costly. Another possibility is to, as in the example illustrated in Figure 6.1, have the memory statically segmented but whenever is needed by an isolated application, the memory can be dynamically remapped. Finally a possible approach between the two aforementioned could be to statically partition and map the memory to caches, leaving a larger partition that can be dynamically mapped according to the isolated application needs.

Finally, the implementation of this approach would come at the price of complexity

requiring the extension of software and hardware mechanisms (i.e., memory mapping tables), in order to be able to dynamically remap memory to caches. This is a work-in-progress that would be worth further exploring.

Validation of the extended version of MPSoCSim: The extended version of MPSoCSim has been used for the exploration of different deployment strategies presented in this thesis work. The main objective was to validate them and to study how they impact the performance of different introduced indicators in order to compare them on several execution scenarios. Consequently, the presented results are relative and show the comparison between the considered deployment strategies. However, the accuracy of the results provided by the extended version of MPSoCSim presented in Chapter 4, Section 4.4, could be evaluated through the comparison of a hardware implementation in order to show the exact accuracy of the simulator. This evaluation might be addressed in future work.

6.4 Conclusion

In this thesis work the problem of cache-based SCAs on many-core systems has been considered. In this manuscript, different contributions on cache-based countermeasures as well as on virtual prototyping tools for multi and many-core systems have been presented. Specially, the spatial isolation of sensitive applications against cache-based attacks has been proposed. This chapter concludes this manuscript discussing first the studied spatial isolation approach in terms of state-of-the-art comparison, then in terms of limitations and possible improvements. Finally, different leads for future work, some of them currently explored, have been described.

This thesis work was realized in the frame of the TSUNAMY project [7] number ANR-13-INSE-0002-02 supported by the French Agence Nationale de la Recherche.

Glossary

ANR Agence Nationale de la Recherche. 3

BTB Branch Target Buffer. 17, 24

CABA Cycle-Accurate-Bit-Accurate. 7

DHCCP Hybrid Cache Coherence Protocol. 7

DoS Denial of Service. 9–11, 28, 38, 119

DQDT Distributed Quaternary Decision Tree. 113

HPC High Performance Computing. 3

ISA Instruction Set Architecture. 27, 28

LLC Last Level Cache. 6, 118, 122

LRU Last Recently Used. 31

LUT Look-Up-Table. 38

LUTs Look-Up-Tables. 38

MMU Memory Management Unit. 6, 10, 12, 33, 38, 40, 119

MPSoCs Multiprocessor Systems-on-Chip. 72

MPU Memory Protected Unit. 10, 33, 40, 119

NI Network Interface. 75, 76, 82

NoC Network-on-Chip. ii, v, vi, ix, 6, 13, 17, 19, 29, 30, 39, 40, 71–76, 78–80, 89, 90, 94, 95, 118, 121, 122

NUMA Non-Uniform Memory Access. 7

OS Operating System. 4, 26, 28, 34–37, 42, 116

OVP Open Virtual Platforms. ii, vi, 71–76, 81, 83, 85, 88, 91, 112, 116

PE Processing Element. 6

PEs Processing Elements. 6

SCAs Side-Channel Attacks. 3, 10, 23, 24, 26, 27, 29, 34, 40–42, 113, 115, 116, 119, 125

SMT Simultaneous Multi-Threading. 26

SZ Secure Zone. 100–102

TCB Trusting Computed Base. 8, 11, 12

TDP Thermal Design Power. 2

TEEs Trusted Execution Environments. 35

TLB Translation Look-aside Buffer. 6, 17, 24

VM Virtual Machine. 5, 12, 24–26, 28

VMs Virtual Machines. 3–5, 12, 13, 17, 25, 28, 37

List of publications and presentations

Publications

Prototyping tools

- Maria Méndez Real *et al.* "MPSoCSim: An extended OVP Simulator for Modeling and Evaluation of Network-on-Chip based heterogeneous MPSoCs". In: Proceedings of the 3th Workshop on Virtual Prototyping of Parallel and Embedded Systems (ViPES) as part of the International Conference on Embedded Computer Systems:Architectures, Modeling, and Simulation (SAMOS). IEEE, 2016. [Paper]

Isolation of sensitive applications on many-core architectures

- Maria Méndez Real *et al.* "Application Deployment Strategies for Spatial Isolation on Many-core Accelerators", submitted to ACM Transactions on Embedded Computing Systems (TECS), 2017, under revision.

- Maria Méndez Real *et al.* "Dynamic Spatially Isolated Secure Zones for NoC-based Many-core Accelerators". In: Proceedings of the 11th International Workshop on Reconfigurable Communicationcentric Systems-on-Chip (ReCoSoC). IEEE, 2016. [Paper]

- Maria Méndez Real *et al.* "ALMOS many-core operating system extension with new secure-aware mechanisms for dynamic creation of secure zones". In: Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and

Network-Based Processing (PDP). Euromicro, 2016. [Paper]

- Maria Méndez Real, Vianney Lapotre, and Guy Gogniat. "Physical Isolation against cache-based Side-Channel Attacks on NoC-based architectures". In: Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS), 2016. [Poster]

- Maria Méndez Real *et al.* "Applications security in manycore platform, from operating system to hypervisor: how to build a chain of trust". In: Springer Workshop of Cryptographic Hardware and Embedded Systems (CHES), available at https://www.tsunamy.fr/trac/tsunamy/wiki/Communications. Springer, 2015. [Poster]

- Maria Méndez Real, Adel Baganne, and Guy Gogniat. "Secure deployment in trusted many-core architectures". In: Proceedings of the 21st IEEE International conference on Electronics Circuits and Systems (ICECS), Women in CAS/Young Professionals/MSc/PhD Forum. IEEE, 2014. [Poster]

Oral presentations

Isolation of sensitive applications on many-core architectures

- Dynamic Spatially Isolated Secure zones for NoC-based Multi and Many-core Accelerators Maria Méndez Real, Vincent Migliore, Vianney Lapotre, Guy Gogniat, - In the International Workshop on Cryptographic Architectures Embedded in Reconfigurable Devices (CryptArchi), La Grande-Motte, France, 2016.

- Special session on the TSUNAMY project Maria Méndez Real, Vianney Lapotre, Guy Gogniat, Mehdi Aichouch, Moha Ait Hmid, Cuauhtemoc Mancillas López, Lilian Bossuet, Viktor Fischer - In the International Workshop on Cryptographic Architectures Embedded in Reconfigurable Devices (CryptArchi), Leuven, Belgium, 2015.

Invited talks

Isolation of sensitive applications on many-core architectures

- Investigation on Spatial Isolation against Logical Cache-based Side-Channel Attacks in Multi/Many-Core Architectures Maria Méndez Real - at the final Conference on Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE), Barcelona, Spain, 2016.

- Spatial Isolation against Logical Cache-based Side-Channel Attacks on Multi/Many-Core Architectures Maria Méndez Real - at the Séminaire sécurité des systèmes électroniques embarqués, IRISA-DGA, Rennes, France, 2016.

Bibliography

[1] *Moore's law*. http://download.intel.com/pressroom/kits/events/moores_law_40th/MLTimeline.pdf.

[2] *Kalray's MPPA*. http://www.kalrayinc.com/kalray/products/.

[3] *TILE-Gx36*. http://www.mellanox.com/page/products_dyn?product_family=237.

[4] *TILE-Gx72*. http://www.mellanox.com/page/products_dyn?product_family=238.

[5] *Epiphany*. http://www.adapteva.com/epiphanyiv/.

[6] *Xeon Phi*. http://www.intel.fr/content/www/fr/fr/products/processors/xeon-phi/xeon-phi-processors.html.

[7] *TSUNAMY project*. https://www.tsunamy.fr.

[8] *TSAR many-core architecture*. https://www-soc.lip6.fr/trac/tsar.

[9] Lubos Gaspar *et al.* "HCrypt: A Novel Concept of Crypto-processor with Secured Key Management". In: *Proceedings of the 11th International Workshop on Reconfigurable Communicationcentric Systems-on-Chip (ReCoSoC)*. IEEE, 2010.

[10] Lubos Gaspar *et al.* "HCrypt: A Novel Concept of Crypto-processor with Secured Key Management". In: *Workshop of Premier atelier sur la Sécurité dans les clouds (SEC2)*. 2015.

[11] Ghassan Almaless. "Operating System Design and Implementation for Single-Chip cc-NUMA Many-Core". PhD thesis. Université Pierre Marie Currie (PMC), 2014.

[12] Cuauhtemoc Mancillas López *et al.* "Trusted computing using enhanced manycore architectures with cryptoprocessors". In: *Proceedings of the 22nd IFIP/IEEE International Conference on Very Large Scale Integration, (VLSI-SoC)*. IFIP/IEEE, 2014.

[13] Clément Dévigne *et al.* "Executing secured virtual machines within a manycore architecture". In: *Microprocessors and Microsystems* 48 (2017), pp. 21–35.

[14] Tobias Bjerregaard and Shankar Mahadevan. "A survey of research and practices of Network-on-chip". In: *Journal ACM Computing Surveys (CSUR)* 38.1 (2006), p. 1.

[15] Richard Buchmann *et al.* "Fast cycle accurate simulator to simulate event-driven behavior". In: *Proceedings of the International Conference on Electrical, Electronic and Computer Engineering (ICEEC)*. IEEE, 2004.

[16] Ghassan Almaless and Franck Wajsburt. "Does shared-memory, highly multi-threaded, single-application scale on many-cores". In: *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism*. USENIX, 2012.

[17] Saman Taghavi Zargar, James Joshi, and David Tipper. "A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks". In: *IEEE Communications Surveys & Tutorials* 15.4 (2013).

[18] Niels Provos, Markus Friedl, and Peter Honeyman. "Preventing privilege escalation". In: *Proceedings of the 12th Conference on USENIX Security Simposium*. USENIX, 2003.

[19] Weizhong Qiang andKang Zhang and Hai Jin. "Reducing TCB of Linux Kernel Using User-Space Device Driver". In: *Proceedings of the Internatonal Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer, 2016.

[20] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Proceedings of the International Cryptology Conference (CRYPTO)*. 1999.

[21] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Introduction to Differential Power Analysis". In: *Journal of Cryptographic Engineering* 1.1st (2011), pp. 5–27.

[22] Daniel Genkin, Lev Pachmanov, and Itamar Pipman. "ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs". In: *Proceedings of the International Cryptology Conference*. Springer, 2016.

[23] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA key extraction via low-bandwidth acoustic cryptanalysis". In: *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA)*. Springer, 2014.

[24] Quian Ge *et al.* "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware". In: *Journal of Cryptographic Engineering (Cryptogr Eng)* (2016), pp. 1–27.

[25] Onur Acıçmez and Çetin Kaya Koç. "Microarchitectural Attacks and Countermeasures". In: *Cryptographic Engineering* (2009), pp. 475–504.

[26] John Kelsey *et al.* "Side channel cryptanalysis of product ciphers". In: *Proceedings of the 5th European Symposium on Research in Computer Security (LNCS)*. Springer-Verlag, 1998.

[27] Michael Neve and Jean Pierre Seifert. "Advances on Access-Driven Cache Attacks on AES". In: *Selected Areas in Cryptography* 4356 (2006), pp. 147–162.

[28] Onur Acıiçmez and Çetin Kaya Koç. "Trace-driven cache attacks on AES". In: *Proceedings of the International Conference of Information and Communications Security (ICICS)*. 2006.

[29] Colin Percival. "Cache missing for fun and profit". In: *Proceedings of BSDCan*. 2005.

[30] Fangfei Liu *et al.* "Fast-Level Cache Side-Channel Attacks are Practical". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015.

[31] Yuval Yarom and Katrina Falkner. "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of USENIX Security Symposium*. 2014.

[32] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing and its Application to AES". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015.

[33] Mehmet Kayaalp *et al.* "A High-Resolution Side-Channel Attack on Last-Level Cache". In: *Proceedings of the 53rd Annual Design Automation Conference (DAC)*. ACM, 2016.

[34] Daniel Gruss *et al.* "FLUSH+FLUSH: A Fast and Stealthy Cache Attack". In: *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assesment (DIMVA)*. Springer, 2016.

[35] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games-Bringing Access-Based Cache Attacks on AES to Practice". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2011.

[36] Dmitry Domnitser *et al.* "Flexible Hardware-Managed Isolated Execution: Architecture, Software Support and Applications". In: *IEEE Transactions on Dependable and Secure Computing* PP (2016), p. 1.

[37] Cezar Reinbrecht *et al.* "Side channel attack on NoC-based MPSoCs are practical: NoC Prime+Probe attack". In: *Proceedings of the 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE, 2016.

[38] Johannes Blömer and Volker Krummel. "Analysis of Countermeasures Against Access Driven Cache Attacks on AES". In: *Selected Areas in Cryptography* 4876 (2007), pp. 96–109.

[39] Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall. "Improved trace-driven cache-collision attacks against embedded AES implementations". In: *Workshop on Information Security Applications*. 2010.

[40] Daniel J. Bernstein. *Cache-timing attacks on AES*. Preprint available at https://cr.yp.to/papers.html#cachetiming. The University of Illinois at Chicago, 2005.

[41] Dag A. Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and countermeasures: the case of AES". In: *Proceedings of the RSA Conference Cryptographers Track (CT-RSA)*. 2006.

[42] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cross Processor Cache Attacks". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)*. ACM, 2016.

[43] *Intel Corporation: Intel(R) 64 and IA-32 Architectures Software Developer's Manual*. https://www.intel.com/Assets/en_US/PDF/manual/253667.pdf.

[44] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. "New branch prediction vulnerabilities in openSSL and necessary software countermeasures". In: *Proceedings of the 11th IMA international conference on Cryptography and Coding*. ACM, 2007.

[45] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. "Predicting Secret Keys Via Branch Prediction". In: *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA)*. Springer, 2007.

[46] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. "On the power of simple branch prediction analysis". In: *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 2007.

[47] Maria Méndez Real, Vianney Lapotre, and Guy Gogniat. "Physical Isolation against cache-based Side-Channel Attacks on NoC-based architectures". In: *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS) as Poster presentation*. 2016.

[48] Maria Méndez Real, Adel Baganne, and Guy Gogniat. "Secure deployment in trusted many-core architectures". In: *Proceedings of the 21st IEEE International conference on Electronics Circuits and Systems (ICECS), Women in CAS/ Young Professionals/ MSc/ PhD Forum*. IEEE, 2014.

[49] Maria Méndez Real *et al.* "Applications security in manycore platform, from operating system to hypervisor: how to build a chain of trust". In: *Springer Workshop pf Cryptographic Hardware and Embedded Systems (CHES), as Poster presentation available at https://www.tsunamy.fr/trac/tsunamy/wiki/Communications*. Springer, 2015.

[50] Maria Méndez Real *et al.* "ALMOS many-core operating system extension with new secure-aware mechanisms for dynamic creation of secure zones". In: *Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. Euromicro, 2016.

[51] Maria Méndez Real *et al.* "Dynamic Spatially Isolated Secure Zones for NoC-based Many-core Accelerators". In: *Proceedings of the 11th International Workshop on Reconfigurable Communicationcentric Systems-on-Chip (ReCoSoC)*. IEEE, 2016.

[52] *International Workshops on Cryptographic architectures embedded in logic devices*. https://labh-curien.univ-st-etienne.fr/cryptarchi/, slides available at https://www.tsunamy.fr/trac/tsunamy/wiki/Communications.

[53] *DGA-IRISA national security seminar*. http://securite-elec.irisa.fr/.

[54] *Final Conference on Trustworthy Manufacturing and Utilization of Secure Devices*. https://trudevice2016.eel.upc.edu/en.

[55] Maria Méndez Real *et al.* "MPSoCSim: An extended OVP Simulator for Modeling and Evaluation of Network-on-Chip based heterogeneous MPSoCs". In: *Proceedings of the 3th Workshop on Virtual Prototyping of Parallel and Embedded Systems (ViPES) as part of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 2016.

[56] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES-The Advanced Encryption Standard*. Vol. 1, Information Security and Cryptography. Springer-Verlag Berlin Heidelberg, 2002.

[57] Michael Godfrey and Mohammad Zulkernine. "A server-side solution to cache-based side-channel attacks in the cloud". In: *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2013.

[58] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. "Scheduler-based Defenses against Cross-VM Side-channels". In: *Proceedings of the Proceedings of USENIX Security Symposium*. 2014.

[59] Yingian Zhang and Michael K. Reiter. "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2013.

[60] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. "New Results on Instruction Cache Attacks". In: *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2010.

[61] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. "Eliminating Fine Grained Timers in Xen". In: *Proceedings of the 3rd ACM workshop on Cloud computing security workshop (CCW)*. ACM, 2011.

[62] Yingian Zhang *et al.* "Cross-VM Side Channels and Their Use to Extract Private Keys". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[63] Juan Campo. "Formally verified countermeasures against cache based attacks in virtualization platforms". PhD thesis. Montevideo : UR.FI.INCO, 2016.

[64] Gilles Barthe *et al.* "System-level non-interference for constant-time cryptography". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[65] Bart Coppens *et al.* "Practical mitigations for timing-based side-channel attacks on modern x86 processors". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2009.

[66] *Constant-time verification tool*. https://github.com/agl/ctgrind.

[67] Dan Page. *Partitioned Cache Architecture as a Side-Channel Defense Mechanism*. Cryptology ePrint Archive, Report 280 https://eprint.iacr.org/2005/280.pdf. 2005.

[68] Benjamin A. Braun, Suman Jana, and Dan Boneh. *Robust and efficient Elimination of Cache and Timing Side Channels*. Preprint arXiv preprint arXiv:1506.00189 available at https://arxiv.org/abs/1506.00189. Standfort University, 2015.

[69] Zhenghong Wang and Ruby B. Lee. "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks". In: *Proceedings of the IEEE Symposium on Computer Architecture (ISCA)*. IEEE, 2007.

[70] Jicheng Shi *et al.* "Limiting Cache-based Side-Channel in Multi-tenant Cloud using Dynamic Page Coloring". In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2011.

[71] Paul Barham *et al.* "Xen and the art of virtualization". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP)*. ACM, 2003.

[72] Wei-Ming Hu. "Reducing timing channels with fuzzy time". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 1991.

[73] Martha Johanna Sepulveda *et al.* "NoC-Based Protection for SoC Time-Driven Attacks". In: *Embedded Systems Letters* 7.1st (2015), pp. 7–10.

[74] David Cock *et al.* "The Last Mile: An Empirical Study of Timing Channels on seL4". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[75] *M-Sim*. https://www.cs.binghamton.edu/~msim/.

[76] Fangfei Liu and Ruby B. Lee. "Random Fill Cache Architecture". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2014.

[77] *Intel Trusted Platform Module Hardware User's Guide*. https://www.intel.com/content/dam/support/us/en/documents/motherboards/server/sb/g21682003_tpm_hwug.pdf.

[78] *Intel Trusted Execution Technology*. https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf.

[79] Ittai Anati *et al.* "Innovative technology for CPU based attestation and sealing". In: *Proceedings of the 2nd international workshop on Hardware and Architectural support for Security and Privacy (HASP)*. ACM, 2013.

[80] *Secure Virtual Machine Architecture Reference Manual*. https://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf.

[81] *ARM TrustZone Technology*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/ch03s03s02.html,ARM architecture, TrustZone.

[82] Moritz Lipp *et al.* "ARMageddon: Cache attacks on mobile devices". In: *Proceedings of the 25th USENIX Security Symposium*. USENIX, 2016.

[83] Hiroaki Inoue *et al.* "Dynamic security domain scaling on symmetric multiprocessors for future high-end embedded systems". In: *Proceedings of the 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM/ACM/IFIP, 2007.

[84] Ramya Jayaram Masti *et al. Isolated Execution in Many-core Architectures*. Cryptology ePrint Archive, Report 280 https://eprint.iacr.org/2014/136.pdf. 2014.

[85] *SCC External Architecture Specification*. https://communities.intel.com/docs/DOC-5044.

[86] Andreas Weichslgartner *et al.* "Design-Time/Run-Time Mapping of Security-Critical Applications in Heterogeneous MPSoCs". In: *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2016.

[87] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. "Prospector: A dynamic data-dependence profiler to help parallel programming". In: *Proceedings of the USENIX workshop on Hot Topics in parallelism (HotPar)*. USENIX, 2010.

[88] Zheng Wang, Georgios Tournavitis, and Bjorn Franke. "NoC-Based Protection for SoC Time-Driven Attacks". In: *Embedded Systems Letters* 11.1st (2014).

[89] *Open Virtual Platforms Imperas Software Limited, "OVP & SystemC"*. http://www.ovpworld.org/technology_systemc.

[90] Philipp Wehner, Jens Rettowski, and Diana Goehringer. "MPSoCSim: An extended OVP Simulator for Modeling and Evaluation of Network-on-Chip based heterogeneous MPSoCs". In: *Proceedings of the 3th Workshop on Virtual Prototyping of Parallel and Embedded Systems (ViPES) as part of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 2015.

[91] Nathan Binkert *et al.* "The GEM5 Simulator". In: *Computer Architecture News* 39.2 (2011), pp. 1–7.

[92] Rodrigo Cadore Cataldo. "Design and Exploration of 3D MPSoCs with on-Chip Cache Support". MA thesis. Porto Alegre, Brazil: Computer Science at Pontificia Universidade Católica do Rio Grande do Sul, 2015.

[93] Lavina Jain *et al.* "NIRGAM: a simulator for NoC interconnect routing and application modeling". In: *Proceedings of Design, Automation and Test in Europe Conference (DATE)*. ACM, 2007.

[94] *Noxim: Network-on-Chip Simulator*. http://sourceforge.net/projects/noxim/.

[95] Nan Jiang *et al.* "A detailed and flexible cycle-accurate network-on-chip simulator". In: *Proceedings of Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013.

[96] Felipe Rosa *et al.* "Fast Energy Evaluation of Embedded Applications for Many-core Systems". In: *Proceedings of the 24th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 2014.

[97] *Imperas Software Limited, Using OVP Models in SystemC TLM2.0 Platforms*. http://www.ovpworld.org/documents/OVPsimUsingOVPModelsinSystemCTLM2.0Platforms.pdf.

[98] *Zynq-7000 All Programmable SoC*. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/.

[99] Jens Rettowski and Diana Goehringer. "RAR-NoC: A Reconfigurable and Adaptive Routable Network-on-Chip for FPGA-based Multiprocessor Systems". In: *Proceedings of the 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2014.

[100] Yao Wang and G. Edward Suh. "Efficient Timing Channel Protection for On-Chip Networks". In: *Proceedings of the 6th IEEE/ACM International Symposium on Networks on Chip (NoCS)*. IEEE/ACM, 2012.

[101] Cezar Reinbrecht *et al.* "Gossip NoC – Avoiding Timing Side-Channel Attacks through Traffic Management". In: *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE/ACM, 2016.

Spatial Isolation against Logical Cache-based Side-Channel Attacks in Many-Core Architectures

Maria Méndez Real

Résumé

L'évolution technologique ainsi que l'augmentation incessante de la puissance de calcul requise par les applications font des architectures "many-core" la nouvelle tendance dans la conception des processeurs. Ces architectures sont composées d'un grand nombre de ressources de calcul (des centaines ou davantage) ce qui offre du parallélisme massif et un niveau de performance très élevé. En effet, les architectures many-core permettent d'exécuter en parallèle un grand nombre d'applications, venant d'origines diverses et de niveaux de sensibilité et de confiance différents, tout en partageant des ressources physiques telles que des ressources de calcul, de mémoire et de communication.

Cependant, ce partage de ressources introduit également des vulnérabilités importantes en termes de sécurité. En particulier, les applications sensibles partageant des mémoires cache avec d'autres applications, potentiellement malveillantes, sont vulnérables à des attaques logiques de type canaux cachés basées sur le cache. Ces attaques, permettent à des applications non privilégiées d'accéder à des informations secrètes sensibles appartenant à d'autres applications et cela malgré des méthodes de partitionnement existantes telles que la protection de la mémoire et la virtualisation.

Alors que d'importants efforts ont été faits afin de développer des contremesures à ces attaques sur des architectures multicoeurs, ces solutions n'ont pas été originellement conçues pour des architectures many-core récemment apparues et nécessitent d'être évaluées et/ou revisitées afin d'être applicables et efficaces pour ces nouvelles technologies.

Dans ce travail de thèse, nous proposons d'étendre les services du système d'exploitation avec des mécanismes de déploiement d'applications et d'allocation de ressources afin de protéger les applications s'exécutant sur des architectures many-core contre les attaques logiques basées sur le cache. Plusieurs stratégies de déploiement sont proposées et comparées à travers différents indicateurs de performance. Ces contributions ont été implémentées et évaluées par prototypage virtuel basé sur SystemC et sur la technologie "Open Virtual Platforms" (OVP).

Abstract

The technological evolution and the increasing application performance demand have made of many-core architectures the new trend in processor design. These architectures are composed of a large number of processing resources (hundreds or more) providing massive parallelism and high performance. Many-core architectures allow indeed a wide number of applications coming from different sources, with a different level of sensitivity and trust, to be executed in parallel, sharing physical resources such as computation, memory and communication infrastructure.

However, this resource sharing introduces important security vulnerabilities. In particular, sensitive applications sharing cache memory with potentially malicious applications are vulnerable to logical cache-based side-channel attacks. These attacks allow an unprivileged application to access sensitive information manipulated by other applications despite partitioning methods such as memory protection and virtualization.

While a lot of efforts on countering these attacks on multi-core architectures have been done, these have not been designed for recently emerged many-core architectures and require to be evaluated, and/or revisited in order to be practical for these new technologies.

In this thesis work, we propose to enhance the operating system services with security-aware application deployment and resource allocation mechanisms in order to protect sensitive applications against cached-based attacks. Different application deployment strategies allowing spatial isolation are proposed and compared in terms of several performance indicators. Our proposal is evaluated through virtual prototyping based on SystemC and Open Virtual Platforms (OVP) technology.

n° ordre: 454

Université de Bretagne-Sud